JSpeed.io: A Javascript Powered Distributed System For High Performance Computing

Victor Santos, Servio Palacios, Ananth Grama

Department of Computer Science, Purdue University

{vsantosu,spalacio,ayg}@purdue.edu

Abstract

As the era of big data becomes mainstream, the need to process large datasets and intensive calculations in a batch and streaming fashion becomes a necessity. We present a new distributed system that offers trivial setup, simple still powerful programming model, and high level of abstraction without sacrificing performance, maintainability, and scalability. We cover how our system compares to similar implementations, its benefits and drawbacks, and our vision of potential applications. We expose how emerging technologies such as Node.js helped us to implementation sandboxing techniques in private, and crowd sourced execution points with high levels of abstraction. Our system has its specific programming model, which involves the implementation of distributed algorithms as finite state machines that switch states asynchronously at each internal or external event. This event-driven automaton programming model makes it possible to create high performance/non-blocking distributed algorithms without the need of any concurrent primitive such as locks, semaphores, and mutexes, and therefore simplifying the implementation and maintainability of the algorithms. We discuss how our system is structured to support multi-tenancy by using JavaScripts higher order functions, closures and callbacks. The system implements cutting edge technologies such as WebSockets and WebRTC to create point to point communication between execution points with low latency and response time. We demonstrate almost linear scaling with the number of execution points (sandboxes) by executing several CPU and I/O bound algorithms such prime numbers calculations and MapReduce. Finally, we discuss different aspects including but not limited to: fault tolerance, security, and how unsolved problems such

as the halting problem becomes a challenge at the moment of implementing a crowd sourced distributed system capable of in-browser execution.

Categories and Subject Descriptors   CR-number [*subcategory*]: third-level

General Terms   Distributed, Performance

Keywords   Event-Driven, Javascript, Cloud Computing, Crowdsourcing

1. Introduction

In present times, there have been a bursting grow big data related terms. From real time analytics[15], to data mining for business, complex datasets classification, and other buzz topics that involves a common element: large scale data processing. Since Google published the MapReduce paper in 2008 [16], researchers and industry have created different kinds of applications [17, 18, 19] on top of this framework to solve problems that are not necessarily suited for the MapReduce. In order to accomplish these solutions, the framework needs to be modified, or find a workaround for dependencies, group communication, or simply an alternative to what MapReduce does not support. All this is done with the sole purpose of getting massive processing speedup in this distributed system at the moment of analyzing gigabytes or petabytes of data. Although there are robust and well known frameworks [14] that can accomplish this, scientists and industry are inclined to use MapReduce based system such as Hadoop[8] to solve distributed and big data problems. Any person with experience in both MPI and Hadoop MapReduce know the difficulty difference at the moment of implementing an algorithm in both platforms. While plain message passing can be extremely flexible, algorithms can become complex and error prone due to synchronization, race conditions, and all standard concurrency primitives. On the other hand, the MapReduce framework takes those drawbacks away by providing a clean and straightforward API that abstract all communication and synchronization details from the user. The main problem with this simplicity and abstraction is that it loses flexibility and adapting several types of algorithms to this framework can be trou-

blesome. In this study, we propose a new distributed system/framework that offers trivial setup, simple still powerful programming model, and high level of abstraction without sacrificing performance, maintainability, and scalability. We are aiming to provide a system with the ability of solving large scale problems in high-end and commodity hardware(even mobile devices) with flexibility, simplicity, yet high performance. The system takes advantage of emerging technologies[3] and the maturity of some others[1] in order to accomplish our target functionality. This paper contains all the details of our early implementation and concepts and is organized as follows:

Section 2 briefly discuss related research and systems. In this section we discuss how existing systems such as SETI@Home [5], CrowdCL [4], River Trail [2], Hadoop [8], and MPI [14] bring together the inspiration and the fundamentals of our system. We present the advantages and disadvantages of each system and how our proposed solution tackles those disadvantages to fit and keep complying with each system original purpose.

Section 3 explains all the implementation details and concepts of our system. This includes:structure, workflow, execution points sandboxing [9,13], fault tolerance, communication, data structures, and the flexible boundaries of the system. This section contains technical details about the implementation engineering and the asynchronous nature of Node.js, and consequently how this influence great part of the system design. We also explain how the previous research [11] contributes to implementing the event-driven concurrency in our system. The system's programming model will be explained in detail in this section. The event-driven finite state machine programming model is one of the most critical concepts of the system. We contrast this model with existing parallel/distributed programming patterns such as message passing[14], and how different algorithms are accomplished with traditional concurrency primitives(locks, mutex, semaphores) and not blocking asynchronous mechanisms(promises, callbacks, signals, and events). Finally, we present the advantages in terms of performance of the JavaScript runtime [1], and how emerging technologies[2, ,20] to reach nearly native execution performance. We also explain how to handle algorithms that are CPU and IO bound in both, Node.js and in-browser environment.

Section 4 presents several experiments that show the performance or our system under different scenarios. From simple number crunching algorithms with no file system requirements, to a MapReduce implementation over Gigabytes of data with heavy I/O operations. We demonstrate how the system behaves in different scenarios, the communication cost, how faults affects the execution, and how to balance sandboxes with different capabilities (from cellphones, to high end servers).

Section 5 explains many important challenges and future work of our contributions. This section discusses the current challenges and what is ahead of the road in the next version of our system. Topics for this section are but not limited to: structural changes and programming model enhancements, fault tolerance or a real time system [7], decentralization and point to point communication [3], code optimization and existing applications porting [6], and the security of the execution points(sandboxes). Finally we briefly discuss the halting problem [10] in our system and the possible solutions [12].

In section 6 we present our conclusions and insights.

2. Design and Implementation

Before we go into the further details of our system design and implementation, its imperative to enumerate several important facts.

Our system does not aim to replace existing solutions but to provide an alternate way to fulfill heavy distributed tasks using a different perspective and approach. The system is based on emerging technologies, some of these are still in development [3] and experimental [4, 2]. Many details in this section are not final and are subject to change as technologies are standardized and improved. As mentioned in the previous section, our system aims to take advantage of crowdsourced computation, commodity hardware, and private dedicated machines for solely distributed processing purposes. We have an experimental prototype of the system working, but not all features are fully implemented yet. Moreover, as this paper is being written, a full restructuration and refactoring of the codebase is taking place to implement all details mentioned ahead.

With the previous facts presented, lets proceed to explore our system in the following eight sections.

2.1 Technologies

Our system heavily relies on relatively recent technologies. All functionalities run on top of any modern ECMAScript engine such as Googles V8 engine, Mozillas SpiderMonkey, Apples Nitro, and Microsoft Chakra. Our system is divided into two kind runtimes: Native: The native runtime uses Node.js V8 engine and has extended capabilities such as disk I/O and full OS interaction capabilities like any other modern language. Browser: Any modern browser following the javaScript standard can become an execution point in our system. However, this runtime is strictly limited by the browsers security policies such as no disk I/O, no direct OS interaction, and limited memory consumption.

javaScript was designed to be a simple scripting language for the web and not designed for performance, but we that is no longer true. javaScript has been the center of countless innovations and new technologies that supports real-time communication (WebSockets). Even the game industry is moving toward browser-based graphics processing (WebGL). In addition, there is a big ongoing effort to make javaScript a high-performance language similar to Java and even C++

[4, 6, 20]. Yet another reason for choosing javaScript as our main language is that tech giants such as Google and Mozilla have created runtimes with remarkable performance and scalability [1].

Several important principles of our system are ruled by javaScripts asynchronous nature. This is, perhaps, the most remarkable difference between our system and existing ones. Let's list the implications this nature on our system:

It does not rely on explicit multithreading to support concurrency at sandbox level, but in subworker processes that run on separated contexts. It relies on a totally asynchronous I/O eventing model. Its event-driven nature offer developers a more efficient and scalable alternative to applications activity switching [1]. Promises easier to implement distributed algorithms, maintainability and execution efficiency over those with explicit multi-threaded implementation [1].

2.2 Programming Model

One of our goals is to reduce the difficulty of plain multithreading programming and its common problems(deadlocks, shared resources collision, and race conditions). With the help of anonymous functions and closures, we maintain the context of all jobs isolated and ready to use at each asynchronous event. In this way, we prevent the use of global data and difficult to maintain data structures. Since the system is totally non-blocking, all I/O is handled asynchronously, therefore, no socket communication, disk IO, or messages from other nodes in the cluster blocks the execution or waste CPU cycles.

2.2.1 The Asynchronous Finite State Machine

The fact that javaScript does not support synchronization primitives, forces us to find an alternative to synchronizing and order algorithms execution flow in the top of its asynchronous nature. This is where we introduce our higher order functions assisted programming model, where all transitions in the algorithm are visualized as an asynchronous finite state machine. In this paper, we will be using the terms event and state interchangeably.
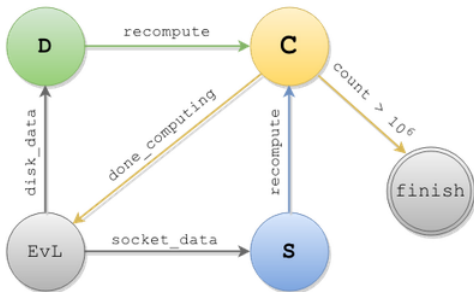

Figure 1. Asynchronous automaton representation of a five state algorithm

Rather than implement threaded procedures with synchronization primitives, distributed algorithms in our system must be represented as a finite state machine of events. In order to illustrate this idea, figure 1 presents a simple distributed algorithm which consists of five states:

Let state an example with figure 1 structure. Given a large dataset containing a sequence of integer numbers, implement an algorithm that reads a section of the dataset from disk and compute the frequency of each integer in the entire dataset. We know that if the dataset contains terabytes of data, we would need to distributively execute this algorithm and receive data from other computing points in our system. In this case, we abstract our problem in states, where state D will asynchronously read from disk, S will receive the pair of ¡integer, frequency¿ list from a network socket, and C will compute/recompute these frequencies with the disk and network data, and finally finish the execution if a certain condition is met. In this previous example, we can notice that our algorithm executes according to events and transitions depending on those events. Moreover, we observe that all the I/O(i.e. disk and socket) is done asynchronously, meanwhile, other states can be executed concurrently. Continuing our previous example, if we go from state EvL to D(data from disk), and at the same time we receive a socket_data event, the internal event loop will take care of executing the queue of events and finally switch state to C where our computation will take place.

Now that we have seen the concept of how our system is meant to execute algorithms, we can make the following observations. We code routines thinking of asynchronous events that will switch the states of our algorithm at any point in time. The runtime itself leverage the synchronization of I/O operations, i.e. we dont need to protect stored data structures from concurrent collisions. Not suspension or blocking of states exist in this system. Instead, each event keeps its own context and when done, the event loop keeps idle or executes the next event. We will encounter situations where we need to manage dependencies in our distributed computation, but we dont have concurrency primitives such as semaphores and locks. We would need to differ asynchronous events if some dependency is not met. In order to do this, we use a combination of constructs (promises, signals, and emitters) which allow us to differ, signal, and emit asynchronous events at will. We briefly discuss how to handle dependencies and ordering of events in section 3.2.2.

Figure 2 shows a slightly more complex scenario. This time we present the asynchronous automaton representation of the Pagerank algorithm.

$$PR(u) = (1-d) + d \sum \frac{PR(v)}{N(v)} \qquad (1)$$

With the left side of the sum constant replaced by d for presentation simplicity:
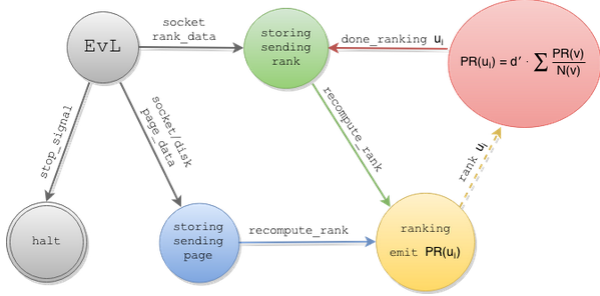
$$d' = (1-d) + d \qquad (2)$$

Figure 2. Asynchronous automaton representation of the PageRank

Figure 3. State dependency scenario

Analogous to the previous example, this automata has the default EvL state, and a termination state called halt, which indicates that this algorithm will halt on signal and not after a certain condition. Its correct to infer that this is a streaming algorithm that will keep computing the received/read page ranks periodically. When a new page is read from disk, or its communicated via the network socket, the corresponding rank of that page is computed and stored asynchronously. Finally, the computed rank(if corresponds to this execution point) will be broadcasted to other execution points(Sandboxes) asynchronously.

2.2.2 Handling state dependencies and ordering

Dabek et. al. states that events are better means of managing I/O concurrency than threads since almost always have latent data races and deadlocks that make them less robust [22]. In addition, they explicitly state that event-driven programs need only to allocate required memory for the events context, and not the whole thread stack, this way, dramatically reducing memory usage. The problem with the event-driven model is that lacks of synchronization primitives to logically state dependencies over asynchronous eventualities. This can be addressed by passing messages between events and used shared structures to differ events and re-order. This way synchronization becomes slightly clearer and reduces the number of global variables in compared to thread-based programs [11].

Lets go back to our figure 1 example and suppose that we have a data dependency between state C(compute) and state D(read from disk). If state S(read from socket) is triggered several times before event D happens, the transition from state S to C cant happen because the data dependency constraint is not met. In this case, we need to implement a different mechanism.

Figure 3 shows the previously exposed scenario where state S cant move to the next state C if at least one event D happens first. We can defer events S using asynchronous event emitter libraries [21] and emit those events when event D happens. After this, our execution continues without problems. In figure 3 we see that S happens three times before D, each time the event is put in a queue to later be rescheduled
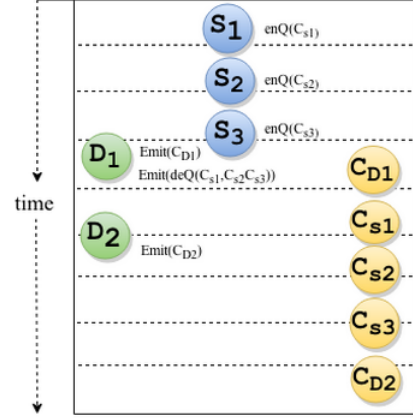
by event D. We have other mechanisms such as promises, callbacks and signals that can be combined to fulfill all kind of dependency and control flow scenarios.

2.3 Architecture

Our system is composed of long-running daemons installed with just one command via the Node Package Manager [26]. Figure 5 shows the peer-to-peer architecture of the system which is comprised of four main components. Each one of the components can be described as follows: Clients: A client(group 2 in figure 5) is the starting point of all executions in the system. This is where we specify our distributed algorithm, the specific Sandboxes to be executed, the data sources, and the different settings of the execution. Each deployment is a job and we can have an arbitrary number of jobs executing at the same time. Sandboxes: Each Sandbox(group 4 in figure 5) is a multi-tenant execution point where all deployed jobs run concurrently. Each time a client deploys a job, all target Sandboxes execute the asynchronous automaton code concurrently. In the next section, we will discuss in detail how the sandbox accomplishes each task. Datasources: The Datasources(group 1 in figure 5) are any remote source that feeds the Jobs Sandboxes. This can be a streaming source, a remote filesystem, or a cloud storage service such as Amazons Cloud Drive[24]. Servers: Each server(group 1 in figure 5) has a register of each Sandbox, Client, and Datasources. The servers monitor every connection to the Sandboxes, clients and Datasources in real-time and notify every relevant change to the corresponding node, i.e. disconnection of a sandbox to an ongoing jobs client. The Server component is inspired by software-defined networking (SDN) Controllers[9].

The system heavily relies on WebSockets. We are currently using the Socket.io [25] library for the real-time communication from which we leverage convenient features such as heartbeats, connection state, and event-driven communication. The Servers are continuously listening for connections from Clients, Sandboxes, and Datasources. Each
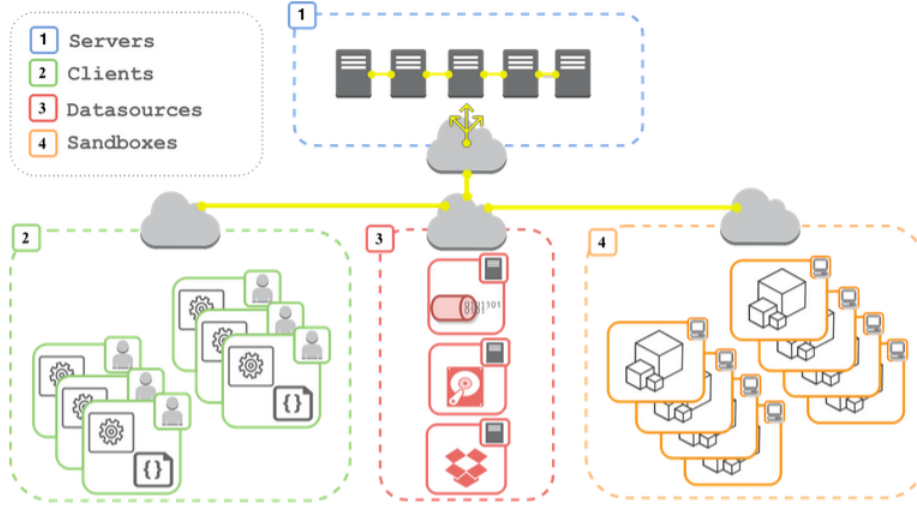
Figure 4. Architecture of JSpeed.io which is comprised of four main components: Servers, Clients, Datasources, and Sandboxes.

time a new Sandbox connection comes in, the servers register the new node and broadcast its presence to Clients. At each disconnection, the Servers also broadcast the disconnection to Clients, in addition to fault tolerance messages if the Sandbox was executing a job.

2.4 The Sandbox execution point

The sandbox is perhaps the most complex component of our system. In its simplest form, the sandbox is a software of contention for executing third party javaScript routines in a distributed fashion. When a Sandbox is executed it follows following tasks: Benchmark the performance of the environment(Node.js or Browser) in order to provide weighting metrics to the clients. These metrics will be used to fairly distribute job's loads among the group of selected Sandboxes at deployment time. Spawn child process pool for jobs execution. Depending on the number of cores of the host, the Sandbox will spawn a subprocess pool where jobs will run. This is necessary since javaScript is single threaded, and each subprocess container will be handle isolated Job execution. Figure 4 shows how the sandbox contains two containers executing asynchronous automatons in an isolated manner. Setup Job's queue structures/registers and inbound/outbound communication(Socket.io events). Notify the Servers and send metrics(benchmark, NAT presence, IP, link speed, runtime capabilities, etc). Wait in the event loop for events (Connection/Disconnection, Jobs). Socket.io takes care of the connection heartbeats for us.

This may look like a straightforward model. In fact, the Node.js version of the Sandbox is simple, efficient and with full capabilities of its environment. We simply spawn the subprocess pool and restrict functionality and resource consumption in those containers. The problem comes when the Sandbox executes in the browser. Since one of the targets of

our system is crowdsourcing and easy setup(not need to install the Node.js Sandbox), is imperative to provide a secure and efficient runtime.
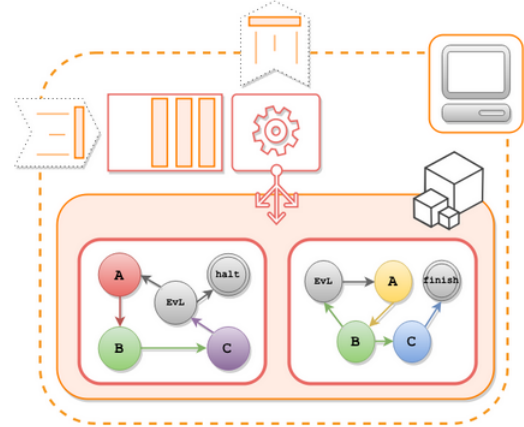

Figure 5. Architecture of the JSpeed.ios sandbox.

Without the proper security and regulations, the system could become the perfect DoS weapon, a free and accessible system with those capabilities could result in disaster. For ensuring complete isolation of deployed Jobs, we recur to the following sandboxing techniques:

HTML5 Web Workers: Web workers are deployed as a background process in the browser and are contained to just use Plain javaScript and XMLHttpRequest, no DOM objects are allowed. The problem with the web workers is that we are executing third party scripts(async automatons) as part of the distributed jobs. With access to do in-domain and different origin requests, we can fall into the mistake of allowing our system to become a DoS attack weapon. Static code analysis can be done to prevent this kind of situations, but the inclusion of libraries can make this process really

difficult, even more in a scripting language like javaScript. JSand [13]: JSand is a sandboxing framework implemented in javaScript that does not require browser modifications. In addition, JSand accesses all resources through the sandbox and enforces server-side policies without rewriting scripts or server-side filtering. Some sandboxing approaches require intrusive browser modifications, others do not support client-side script inclusion which effectively changes the architectural model of client-side script inclusion to server-side. Some approaches do not provide complete mediation between different scripts on the same page, or to all resources exposed in the browser.

Sandboxing techniques can protect the host machines of becoming a DoS and any access to DOM and Window objects. Still, there are other issues which we should be concerned. One of these problems is malicious deployed code that can deplete the host machine resources and even crash it. In section 3.8 we discuss a possible approach to this kind of problem.

2.5 Fault Tolerance

Our system is based on states of change and signals from external sources(peer Sandboxes), therefore, its ruled by the spontaneous initiatives of activities, which results in difficult to handle faults [7]. According to Kopetz et. al. in this type of system we assume that components (Sandboxes, Clients, and Datasources) posses self-checking properties and fail silently. Moreover, they operate as intended, or do not produce any results. Since we have transient components, we would need redundant/synchronized self-checking nodes that as long as one of those components can operate, our required service can be maintained. Our system prototype implements a reschedule-if-fails policy. If we deploy a job with 20 Sandboxes, and 4 of them disconnect or fail, the payload of those 4 Sandboxes are rescheduled and gradually deployed when the first 4 of the 16 currently running finish their task. This without doubt degrades performance and currently only works with algorithms with not group communication requirements. Clearly inefficient, this approach at least guarantees the jobs completion as long as there is one Sandbox available. Fault tolerance will be a major area in the next development stage of the system.

2.6 Limitations

Our system has several limitations that are mainly due to the security and its client-server nature of web technologies. In the following subsections, we briefly discuss the two of this limitation.

2.6.1 Browser Execution

Currently, we can use two execution points: Any browsers ECMAScript runtime, where users go to a web page and automatically setup all the execution environment and a Node.js application that currently runs on top of the V8 engine with full I/O and OS interaction capabilities. The browser runtime is limited to CPU-bound processes since it does not allow access to the file system because of security reasons. For this reason, we create the Datasource component which allows deployed jobs to fetch binary data from remote services, file systems, and streaming APIs. In addition to the previous limitation, we have the problem of not knowing the exact hardware capabilities of the machines. Some browsers support some hosts hardware information such as the number of CPU cores and bit architecture, but this certainly is not enough to precisely measure the power of the processing sandbox. At the setup of each Sandbox, we do a pessimistic approximation of the sandboxes processing power by running a number crunching benchmark and calculating execution time.

2.6.2 The Halting Problem

One drawback we are facing in our system is the halting problem [10]. Its true that this is an unsolved problem that is present in any Turing Machine based computer. The problem is more noticeable in a system like ours where one of the targets is crowdsourcing. Every distributed execution is critical to the overall progress of the algorithm, the question is if one particular sandbox will perpetually execute in an infinite loop or its gradually progressing towards the solution. A possible approach to this problem is the timed automata [12]. If we are running our sandbox in a Node.js environment, we then have the ability to monitor our job process and decide when to terminate if too much memory or CPU cycles are consumed. On the other hand, if we are in a browser, things get much more complicated since we don't have access to low-level OS information such as available memory and process execution. One possible solution is to extend our event-driven finite state machine model to a timed automata. We can extend our automata with real-valued variables in order to logically clock the execution. By using this technique, we can restrict the execution and behavior of our job and force it to switch into a termination state by following a certain policy.

3. Experiments

In order to present empirical evidence of JSpeed.ios effectiveness, we designed two different experiments. First, we implemented the MapReduce algorithm and ran this job 16 times upon a 2GB CSV file. Each time, we incremented the number of native(Node.js) Sandboxes in order to obtain speedup and efficiency data. For this first experiment, we used 16 machines to run the Sandboxes, each machine has a 1.86GHz Quad Core Intel Xeon E5320 with 8GB of RAM running CentOS Linux 64 bits. The second experiment consists of the same MapReduce Job, but this time we measured the scalability of the system as data size increases. We compared Hadoop and JSpeed.io execution time by running 10 iterations incrementing the data size from 1GB to 10GB. This time, we have been running Hadoop and Speed.io us-

ing 10 machines as nodes/Sandboxes. Each machine is powered by a 2.4Ghz Quad Core Intel Xeon X3430 with 8GB of memory running Ubuntu Linux 64 bits.

Our MapReduce algorithm is shown in Listing 1 and Listing 2. The map function takes a CSV line and extracts the first letter of the first column(first name), and calculate the fourth column date difference(age). Finally, it creates an object defined as String: Integer which is a mapping of the first letter and age. The reduce function receives each map object and reduce all objects in a hash of ¡letter,frequency(t),age(c)¿, which is the mapping of the first letter, its frequency and accumulated age.

```
var flds = params.split(",");
var fl = flds[0].charAt(0).toUpperCase();
var d1 = new Date(flds[3]);
var d2 = new Date();
var age = d2.getYear() - d1.getYear();
result = {};
result[fl] = {c:1,t:age};
```

Listing 1. Job's map function code

```
for(var key in now){
  if(last[key]){
    last[key].c++;
    last[key].t += now[key].t;
  }
  else{
    last[key] = now[key];
  }
}
result = last;
```

Listing 2. Job's reduce function code

3.1 SpeedUp and Efficiency

At each deployment, we specified the number of Sandboxes (from 1 to 16), and provide the path of the 2GB CSV along with the total number of lines in the file. If the job will be executed in 2 or more Sandboxes, the file is partitioned according to the power of each Sandbox. After executing the MapReduce job in 1 to 16 Sandboxes, we observed a linear speedup. Figure 7 illustrates the linear speedup, where 2 Sandboxes speed up to 1.94 times faster and 16 Sandboxes up to 10.15 times faster.

In figure 8 we can observe how the efficiency of the execution drops as the number of Sandboxes increases. As more execution points are added, the overhead of communication, disk I/O, and scheduling increases. We observe how the efficiency drops from 97% with 2 Sandboxes, to 63% with 16 Sandboxes. We can observe the biggest drop from 3 to 4 Sandboxes.

We observed significant speedUp and efficiency if we consider the fact that our system still in a prototypical stage and we havent implemented important optimizations and
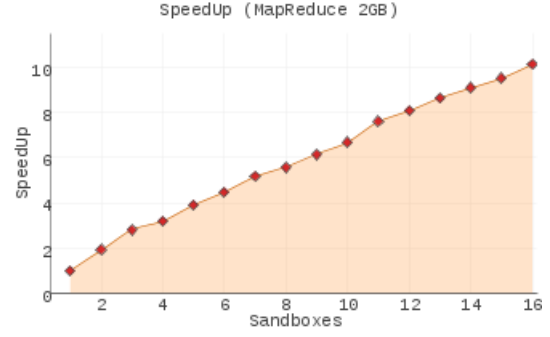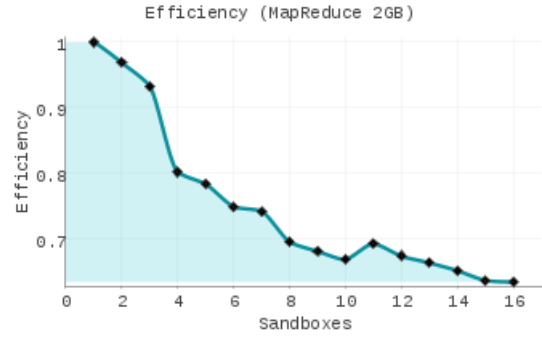

Figure 6. SpeedUp MapReduce with 2GB.

Figure 7. Efficiency MapReduce with 2GB.

tuning. The Server, Client and Sandbox components of this experiment were remotely located, i.e. all communication was over the internet and not in a LAN.
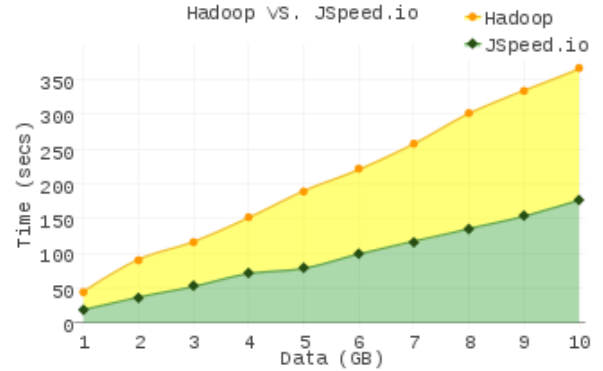
3.2 Scaling

Figure 8. JSpeed.io Vs. Hadoop MapReduce Task.

In our second experiment, we benchmark the scaling of Hadoop MapReduce vs. JSpeed.io implementation as shown in listing 1 and 2. After measuring processing time of all datasets (1GB-10GB) on both systems, we observed slightly different behavior in the linear scaling. As shown in figure 9, JSpeed.io linear execution times grows slower than those in Hadoop. Although our system is not running on a LAN, but over the internet, and each Sandbox is single threaded, it

shows better performance than Hadoop, running on the same machines, with equivalent algorithms, and yielding the same results. Figure 9 shows the difference in completion times on both systems. The 10GB dataset took 367 and 176 seconds to complete with Hadoop and JSpeed.io respectively. With a difference of 197 seconds, JSpeed.io process the biggest dataset 2.09 times faster than Hadoop. On the other hand, Hadoop is way more complete and robust than our system. JSpeed.io currently lacks basic features, such as execution checkpoints, group communication, long studied fault tolerance approach, and many other elements that cause overhead in the overall execution flow. Future work is needed in order to achieve a more fair and accurate comparison between these two systems.

4. Future Work

Although we have presented the potential of this system, you could say that is far from being a complete and robust system. We currently have a long roadmap of features and basic components that must be implemented. Important tasks in our roadmap include but is not limited to:

Structural changes: The system's structure still in its conceptual phase. We need to restructure all components to follow a robust distributed pattern. A major task is to fully integrate WebRTC [3] for dynamic and transient connectivity between Sandboxes, i.e. the dynamic creation of topology overlays on each Jobs deployment. Programming interface: Programming interface must be formalized. This includes the event-driven finite state machine model, single/group communication operations, third party libraries importing, and a clear guide of how to implement synchronization in every possible scenario.

Fault Tolerance: We need to implement two main fault tolerance approaches, one for batch jobs, and another for streaming jobs. Both approaches must deal with the transient nature of our system. i.e. Sandboxes leaving and entering the system at any point of a computation. This is one of the most important and challenging tasks in our roadmap. Optimization and porting: An important task in our future work is to take advantage of several optimization techniques for the deployed jobs code. From static code optimization and transformation to the possibility of porting any language that can compile to LLVM into pure Javascript code [6]. Our system needs to exploit all performance technologies and improvements in order to yield high performance at each distributed execution.

Security: Security and privacy are a concern. We need to do an exhaustive analysis of the different scenarios where our system could become harmful or a threat to the users privacy or system. We will be implementing state of the art and novel approaches to ensure the robustness and fidelity our system.

Additional to all tasks mentioned above, we plan to perform a wider and a detailed comparison between JSpeed.io and current leading systems/frameworks [8, 14]. We will cover important aspects such as power consumption, memory usage, network usage, and failure recovery under different workloads and scenarios.

5. Related Research and Systems

Our goals are analogous to those of volunteer computing projects such as CrowdCL[4], SETI@Home[5], because they work on the premise that users can willingly contribute idle CPU cycles in order to solve large-scale computational problems. In addition, JSpeed.io implements parallelism and sandboxing techniques using a JavaScript based framework similar to River Trail[2] and JSand[13].

MacWilliam et al. achieved the simplified development of web-based GPU applications through a volunteer computing framework called CrowdCL[4]. CrowdCL consists of client and server components that provide an open-source volunteer computing framework. CrowdCLs client framework allows developers to run Javascript code in the background of web browsers. Developers can use the CrowdCLs framework to write OpenCL applications using Javascript, distribute computations to the user's web browser clients, and finally gather results on the server[4]. On the other hand, our framework affords distributed execution of JavaScript implemented kernels using volunteers web browsers.

SETI@home [5] is a scientific experiment that uses Internet-connected computers in the search for extraterrestrial intelligence. The BOINC project provides the framework that powers SETI@home. SETI@home has reached hundreds of TeraFlops from volunteer computing running on desktop applications. Their approach is to divide independent data from observations into small chunks that a personal computer can handle and analyze. In this way, they can distribute the work to people willing to donate their spare CPU cycles [5]. SETI@Home is a native desktop application, in contrast JSpeed.io is based on web technologies capable of executing on both users web browser and native Node.js runtime.

River Trail provides deterministic parallelism to JavaScript applications. Moreover, River Trail is a parallel programming model and API for JavaScript which allows utilization of multiple cores, vector instructions, and GPUs on the user's web browser [2]. Similarly, JSpeed.io utilizes asynchronous parallelization techniques on the sandboxes improving performance. Our system is not an API, but a combination of emerging web technologies that form a distributed system running on top of the JavaScript runtime. In other words, technologies such as WebCL and River Trail can be easily integrated to improve JSpeed.ios performance.

6. Conclusions

We presented the JSpeed.io distributed system. With quick setup, maintainability, scalability, simple programming model, and performance, we aim to provide a powerful and straight-

forward platform to process distributed algorithms in a simpler manner. We presented examples of common algorithms and our vision of potential applications by exposing its benefits and drawbacks. The implemented programming model abstracted as a finite state machine aims to create high performance/non-blocking distributed algorithms without the need of any concurrent primitive such as locks, semaphores, and mutexes.We exposed the linear scaling or our system and how it can be used on different platforms such as web browsers and standalone ECMAScript runtime (Node.js). By easing the implementation, deployment, and maintainability of distributed algorithms, our contributions aims to attract scientists of all fields to immerse into big data processing and distributed computation.

Acknowledgments

References

[1] Tilkov, S., and Vinoski, S. (2010). Node. js: Using JavaScript to build high-performance network programs. IEEE Internet Computing, 14(6), 0080-83.

[2] Herhut, S., Hudson, R. L., Shpeisman, T., and Sreeram, J. (2013, October). River Trail: A path to parallelism in JavaScript. In ACM SIGPLAN Notices (Vol. 48, No. 10, pp. 729-744). ACM.

[3] Vogt, C., Werner, M. J., and Schmidt, T. C. (2013, October). Leveraging WebRTC for P2P content distribution in web browsers. In Network Protocols (ICNP), 2013 21st IEEE International Conference on (pp. 1-2). IEEE.

[4] MacWilliam, T., and Cecka, C. (2013, September). CrowdCL: Web-based volunteer computing with WebCL. In High Performance Extreme Computing Conference (HPEC), 2013 IEEE (pp. 1-6). IEEE.

[5] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Lebofsky, M. (2001). SETI@ HOMEmassively distributed computing for SETI. Computing in science and engineering, 3(1), 78-83.

[6] Zakai, A. (2011, October). Emscripten: an LLVM-to-JavaScript compiler. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (pp. 301-312). ACM.

[7] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. (1989). Distributed fault-tolerant real-time systems: The Mars approach. Micro, IEEE, 9(1), 25-40.

[8] Hadoop. URL: http://hadoop.apache.org.

[9] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

[10] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. J. of Math, 58(345-363), 5.

[11] Haller, P., and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science, 410(2), 202-220.

[12] Alur, R., and Dill, D. L. (1994). A theory of timed automata. Theoretical computer science, 126(2), 183-235.

[13] Agten, P., Van Acker, S., Brondsema, Y., Phung, P. H., Desmet, L., and Piessens, F. (2012, December). JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In Proceedings of the 28th Annual Computer Security Applications Conference (pp. 1-10). ACM.

[14] Gropp, W., Lusk, E., and Skjellum, A. (1999). Using MPI: portable parallel programming with the message-passing interface (Vol. 1). MIT press.

[15] Chen, H., Chiang, R. H., and Storey, V. C. (2012). Business Intelligence and Analytics: From Big Data to Big Impact. MIS quarterly, 36(4), 1165-1188.

[16] Dean, J., and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[17] Chu, C., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G., Ng, A. Y., and Olukotun, K. (2007). Map-reduce for machine learning on multicore. Advances in neural information processing systems, 19, 281.

[18] Ekanayake, J., Pallickara, S., and Fox, G. (2008, December). Mapreduce for data intensive scientific analyses. In eScience, 2008. eScience'08. IEEE Fourth International Conference on (pp. 277-284). IEEE.

[19] Papadimitriou, S., and Sun, J. (2008, December). Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. InData Mining, 2008. ICDM'08. Eighth IEEE International Conference on (pp. 512-521). IEEE.

[20] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., and Fullagar, N. (2009, May). Native client: A sandbox for portable, untrusted x86 native code. In Security and Privacy, 2009 30th IEEE Symposium on (pp. 79-93). IEEE.

[21] Async. URL: github.com/caolan/async

[22] Dabek, F., Zeldovich, N., Kaashoek, F., Mazires, D., and Morris, R. (2002, July). Event-driven programming for robust software. In Proceedings of the 10th workshop on ACM SIGOPS European workshop (pp. 186-189). ACM.

[23] Amazon Cloud Drive, URL: https://www.amazon.com/clouddrive/home

[24] Socket.io, URL: http://socket.io

[25] Node Package Manager, URL: https://www.npmjs.com