RASSMon: Realtime Asynchronous Service Status Monitoring

Bianca Neagu[1], Corina Dulea[2] and Horia V. Corcalciuc[1]
[1]Department of Computational Physics and Information Technologies (DFCTI)
[2]Nuclear Training Centre (CPSDN)
Horia Hulubei National Institute for R&D in Physics and Nuclear Engineering (IFIN-HH)
Magurele, Romania

***Abstract*— The operation and maintenance of large distributed systems that are subject to high QoS conditions has led to the need of designing and developing advanced monitoring tools that facilitate the administration of the critical services required by the user communities. RASSMon is a portable, reliable, secure software platform able to collect monitoring data from multiple sources in heterogeneous environments that offers an unified overview on the resource and service status in distributed systems such as computing grids. This article presents the challenges encountered in order to create a real-time management tool for monitoring grid services, with reference to some of the latest theoretical advances in programming language semantics.**

Keywords: monitoring, grid computing, distributed computing, computer networks.

I. Introduction

The monitoring of the service availability in distributed systems of resources can be accomplished in two ways: (a) by monitoring agents that run locally, or (b) by launching test jobs remotely. When the infrastructure is dedicated to the support of the public scientific research, the fulfilment of the Service Level Agreement (SLA) by the service providers is checked through monitoring tools that are usually under the control of the service consumers. The consumers prefer that the SLA be monitored through method (b) but then, more than often, the resource/service providers have no access to the means for locally reproducing the consumer monitoring tests and only have access to some external (web-)published monitoring reports. This represents a major drawback for providers, whose reaction speed to service incidents becomes heavily dependent on the human interaction with the external monitoring system, if independent availability tests of the aforementioned services are not possible locally.

Such a situation is met, for instance, in the case of the Worldwide LHC Computing Grid (WLCG) [1], where the monitoring of the availability of the experiment-specific services is performed by CERN [2] and cannot be reproduced by the local resource providers. Furthermore, grid jobs are submitted asynchronously without previous knowledge of when a specific task will be processed or even to which machine a task will be delegated to. In turn, this poses a challenge for system administrators, that must make sure that all the resources they maintain offer services with very high availability [3].

The administration of machines that contribute to the grid is left up to the resource centers without imposing a given set of tools but requiring that a certain computational quota / SLA is respected [4]. System administrators have been using a range of applications such as Check MK [5] or Nagios [6] in order to monitor uptime and ensure the availability of services. Unfortunately, the tools being used by administrators for grid monitoring have a broad context, have numerous usage cases and are not tailored specifically to particular requirements, such that mundane tasks (generating specific reports, tracking the amount of downtime, etc.) become tedious and tend to require human intervention.

When the jobs are distributed globally to systems world-wide, an additional difficulty is that local administrators are not able to query the monitoring status provided by global tracker in order to be able to correlate the system status with the local monitoring. One example thereof is the ETF [7] website at CERN, that provides a Check MK instance reporting on the status of the WLCG sites. Unfortunately, the ETF Check MK interface does not provide a means to extract the information conveniently and is meant as a display-only report.

Since various resource centers either allow restricted or partial access to monitoring data without the ability to retrieve and then process it automatically, it becomes clear that some software must be designed to address the shortcomings and allow an unified overview of all the remote sites that system administrators must maintain. Based on user-experience pertaining to grid monitoring this article exploits the latest state of the art programming semantics and technologies with the hope of creating a reliable monitoring platform that is able to harness data from multiple grid-related sources in a heterogeneous environment.

II. Overview

The "Background" section III presents the context of the paper to the reader, explains the main design decisions that were taken during the development of RASSMon and introduces various technical concepts.

In the "Software Architecture" section IV, the inner-workings and design of RASSMon is explained with special emphasis being placed on the interaction between the backend

This work was partly funded by the Ministry of Research and Innovation under the contracts no. 6/2016 (PNIII-5.2-CERN-RO), PN16420202/2016, and from the Hulubei-Meshcheryakov collaboration, project No. 94/2017.

and frontend component of the project. Details are given on the configuration file that RASSMon uses, the necessity to use standards-compliant technologies and various security-related aspects of the developed software package are presented in the "Security" subsection IV-C.

The "High Performance using Streams, Events and Asynchronicity" section V section presents the results of various comparative benchmarks between the HTTP server embedded in RASSMon and Apache whilst attempting to explain why under certain circumstances RASSMon outperforms Apache.

The "Collecting, Serving and Processing Data" section VI explains the flow of data between the two components of RASSMon - from the initial retrieval from CERN and up to displaying the aggregated data to a client.

The usage of third-party code is via package managers is explained in "Packaging and Distribution" section VII along with the ongoing plan to create an official release of RASSMon.

Finally, the conclusions are summarized in section VIII with pointers to further development of the RASSMon package.

III. Background

Most of the encountered grid-monitoring technologies either use a general-purpose platform such as Check MK or borrow and incorporate traditional Unix tools and scripting languages such as Bash or Python [8], in order to do the post-processing of the acquired data. Whilst broad-range platforms suffer from the lack of customizability and would require a fork of the source-code, home-brew options are designed to work in a specific environment. On the other hand, designing C/C++ applications with low-level memory management could potentially have serious security implications for which the risks would not outweigh the benefits given the context of a world-wide computational grid. In fact, due to C and C++'s direct access to memory and the ensuing inherent insecurity, efforts have been made previously to derive the language and introduce safe constructs [9] with a great deal of research done in the area of concurrent programming [10]. Whilst low-level programming remains the only option for designing hardware interfaces, design patterns have shifted from traditional imperative programming semantics to functional programming [11] where the main concern of the programmer is to reason about the programming logic than to deal with platform-specific instrumentation.

One of the better software design patterns is to reuse existing protocols and standardized formats [12]. A lot of the backoffice processing of data for grid services written in shell languages do not adopt nor adhere to any standard; data may be filtered using regular expressions but it is never offered in a standardized language such that it could be parsed. Check MK does offer comma-separated values (CSV) as an export option but the resulting data is not compliant with any standard such as the one specified in RFC 4180 [13].

The main goal of this research is to produce a tool that must be user-friendly and provide easy access to data without requiring advanced knowledge of platform specific tools - most of the operators should not bother with the technicalities but have to be able to report service state changes. The developed utility must provide an user-interface that is capable and reliable enough to offer a local and grid-wide site overview. Additionally, the software should be aware of the inherent heterogeneous environment of grid sites and be easily deployed.

IV. Software Architecture

We leverage concepts from asynchronous programming, choosing NodeJS [14] as the engine for the backend server responsible for doing the heavy-lifting and then provide a web-based interface using state of the art tools such as the NodeJS package manager itself and Twitter Bootstrap [15] for the client experience. During the design phase of RASSMon [16], a minimalistic approach was chosen such that all the code is written in JavaScript and uses only HTML / CSS as markup and styling for the interface. The design goal has been to minimize requirements and to re-use existing technologies and protocols in order to create an application that could be accessed on all platforms.

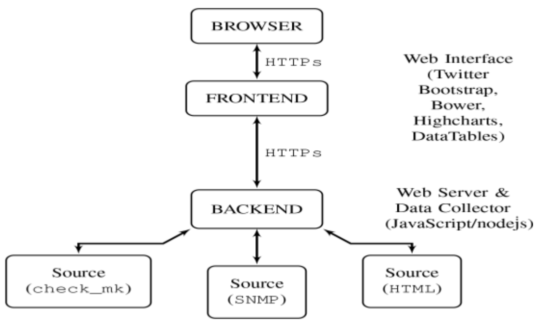
BROWSER
HTTPs
FRONTEND
HTTPs
BACKEND
Web Interface (Twitter Bootstrap, Bower, Highcharts, DataTables)
Web Server & Data Collector (JavaScript/nodejs)
Source (check_mk)
Source (SNMP)
Source (HTML)

Fig. 1. RASSMon provides an user-interface to connecting clients via a built-in HTTP server and communicates with data sources using various protocols that are implemented by the backend.

The architecture of the project illustrated in Fig. 1, is designed with client usability in mind, benefiting from a frontend that serves web-pages and a backend that is responsible for harvesting data from multiple sources based on JavaScript plugins. The plugins for RASSMon are written in JavaScript, imported into the backend at run-time as a NodeJS module and are responsible for pulling and storing data into the frontend website filesystem tree. The Check MK plugin is used to query the data provided by the CERN ETF platform website in CSV format and to serialize the obtained data to JavaScript Object Notation (JSON) [17] files that are later retrieved on access by the frontend when accessed from a web browser. RASSMon stores data to files instead of databases since only a fixed time-slice has to be observable without the need to maintain a history of service state changes. The NodeJS backend permutes files at given intervals, discarding older data files that are not within the configured time period.

A. *Configuration*

RASSMon is configurable through a YAML [18] configuration file that is processed by the NodeJS server backend and by the website-based frontend. The configuration allows adding servers to be monitored dynamically such that the addition or removal of a server from the configuration file will make both the frontend and backend adapt and generate charts and tables without having to alter code. Extending or shrinking the time interval to be monitored is configurable dynamically and only requires alterations to the YAML configuration file. YAML has been chosen due to its flexibility and readability as well as its major advantage over JSON to support comments within the configuration file.

The metrics are generated in real-time such that a client viewing the web-page will notice whether a service or a server has changed state. In case there is no client connected, the backend will still poll the sources in the background and perform any required automated tasks.

B. *Use of Technologies and Standards Compliance*

On a lower level, the project re-uses existing technologies, protocols and established standards - for instance by using the already-provided NodeJS low-level socket implementation, using standardized languages such as JSON or YAML, securing the communication between the browser and the backend using HTTPs, using log4j [19] compatible logging and so forth. The main goal during the development of RASSMon has been to reuse standards-compliant code instead of re-implementing boilerplate requirements. For instance, NodeJS does implement a module that can serve HTTP requests up to the point of creating request and response JavaScript objects but it is up to the programmer to process a request in a meaningful and useful way.

As far as HTTP is concerned, NodeJS does not explicitly implement advanced features such as caching, nor does it NodeJS explicitly distinguish between HTTP1.0 or HTTP1.1 requests. Instead NodeJS provides the necessary API components for programmers to implement the protocol in a standards-compliant way that will be compatible across different browsers. RASSMon uses the NodeJS and other third-party package to implement a minimalistic web-server. The web-server implemented by RASSMon is capable of serving static files by responding to HTTP GET requests, generating directory listings for directories without a default document in JSON format (contrary to Apache that generates HTML-based directory listings) for easier interaction with the web-based frontend, implements HTTP1.1 caching via ETags for all served static files, and additionally serves the RASSMon configuration file when the browser requests a special pre-defined path.

C. *Security*

RASSMon serves all pages through HTTPs and additionally has the ability to request authentication through the HTTP digest authentication scheme. The HTTPs certificates for RASSMon can be generated using provided scripts based on a provided certificate template. A HTTP digest scheme authentication scheme has been added in order to restrict access and only permit known clients from accessing the project.

The distinction between the frontend and the backend allows for some privilege separation where clients are able to only read data and not interact with the running daemon which is highly desirable because the attack surface [20] can be minimized. All browser requests are bound to the document root of the frontend and checked for path-traversal attempts, effectively locking clients into root-jail, thereby prohibiting access to system files.

Further developments may introduce some form of Access Control Lists (ACL) [21] for the interface in order to be able to change server parameters.

V. High Performance using Streams, Events and Asynchronicity

One of the highlights during the development of RASSMon has been to maximize the performance due to the need to issue a large number of requests to retrieve data as well as serve frontend content to connecting clients simultaneously. Whilst low-level programming languages provide threaded execution, NodeJS and JavaScript provide single-threaded asynchronous semantics. The code leverages lambda calculus semantics with lazy evaluation [22] and the environment performs the lambda applications when convenient for the scheduler whilst returning awaitable promises [23] as part of the postcondition of function calls. Using asynchronous execution, the software is able to query multiple sources and to serve pages without having to wait for each code segment to run.

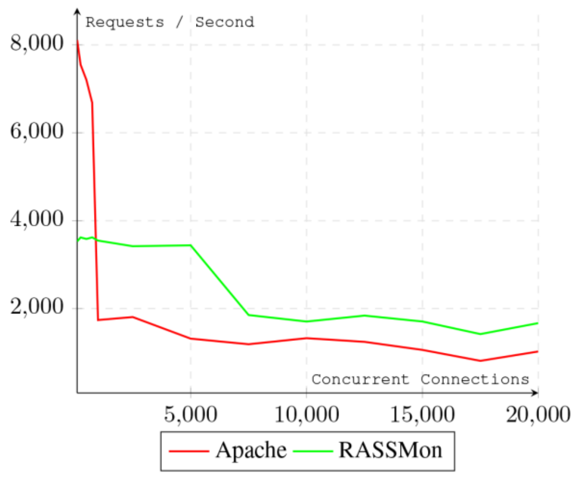

Fig. 2. Throughput plot between the number of concurrent connections to each server versus the number of requests per second that the server responded to.

Fig. 2 is a throughput comparison between an Apache/2.4.23 using the standard prefork module shipped with the latest stable Debian distribution and the RASSMon backend HTTP server. The benchmark was performed on cold-start, with multiple trials, by sending a total of 50000 GET requests and increasing the number of concurrent connections on every iteration.

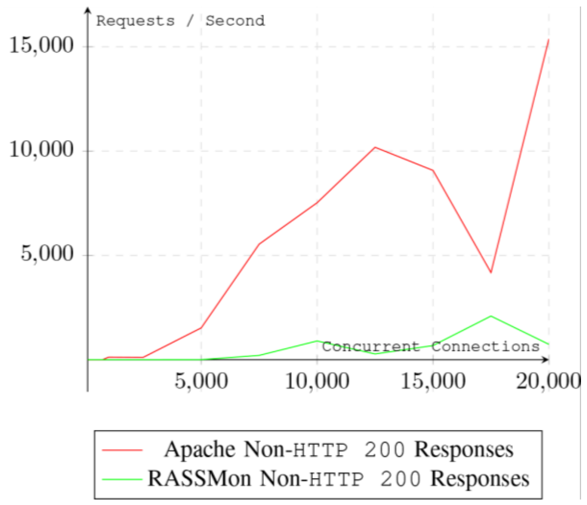

Fig. 3. The plot of non-HTTP 200 responses (failures) comparing Apache and RASSMon whilst the number of concurrent connections are increased during each trial.

As the results show, for a small amount of concurrent requests, Apache tends to take the lead but after 1000 simultaneous requests, RASSMon outperforms Apache. From the gathered data during the benchmarks, a trend may be observed by comparing the number of non-HTTP 200 replies (failed requests) in Fig. 3 indicating that whilst RASSMon does see an increased of failed requests after 7500 concurrent connections, Apache takes the lead very early on after only 1000 requests. Towards the end of the benchmark where 20000 simultaneous requests are sent, Apache tends to fail to respond with HTTP 200 whereas RASSMon is still answering and not failing much more than it did when 10000 requests were being made. Neither Apache nor RASSMon have been altered nor configured specifically for performance - in fact, during the benchmark RASSMon had not been configured to cache static files whereas Apache does. During the benchmark, Apache was solely tasked with serving static files but RASSMon had to additionally gather data from external sources.

The results could be explained due to Apache spending a significant amount of time forking children to process the requests, whilst RASSMon wastes no time on system calls or context switches but rather interleaves the requests and serves the responses when data is made available. Similarly, the high rate of failure of Apache versus RASSMon could perhaps be attributed to low-level errors that become more apparent under a high load -errors that are handled semantically and processed in RASSMon using continuations such as exceptions without burdening other concurrent requests. An Apache process or a thread (for threaded Apache workers) serving multiple clients that encounters a serious error might terminate and consequently drop the connection for other clients served by the same process or thread.

In order to query the service status of multiple grid servers, RASSMon must first obtain a list of servers from the ETF site and then filter them through the selected servers in the configuration file. Once the servers have been filtered, each server must be queried for the status of its services. In NodeJS, all operations run asynchronously on the same thread, with the distinction that the software will await the completion of all tasks, leaving it up to the environment to interleave the execution at scheduled time.

The main task of the CERN ETF plugin of RASSMon is to poll services and store the result to files for processing by the frontend such that sequential execution would have taken an unacceptable amount of time. Operations that need linearity can be awaited upon using various language semantics such as JavaScript promises and signaling completion by emitting events as well as making use of data streams [24, 25] for synchronization. It becomes a challenge to distinguish between operations that require linearity and operations that can be scheduled ad-hoc regardless of the state of the environment.

Whilst threads provide a more context-oriented level of separation and execution, the asynchronicity required by RASSMon does not need the additional overhead incurred by threading or forking. Counter-intuitively, even with the lack of threads, NodeJS has outperformed Apache and PHP when serving dynamic content [26].

VI. Collecting, Serving and Processing Data

The data gathered by the NodeJS backend is stored into the document root of the frontend (Fig. 4). Since the requirements of the project do not imply archival, the stored data has a limited lifetime that is defined in the YAML configuration file and all other previously retrieved data is discarded automatically by RASSMon. In case further developments would require the storage of the retrieved data for a longer period, then software packages such as Web SQL Database (WebSQL) [27] could be used in order to maintain the portability of the project.

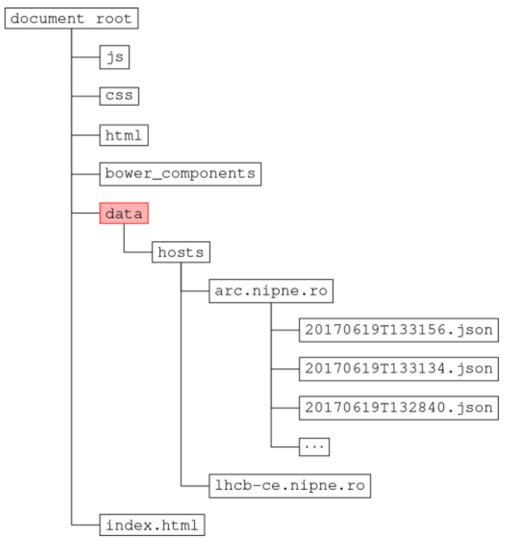

Fig. 4. Filesystem layout of the frontend serving pages to browsers. The data folder is populated by the backend by storing JSON files for each host specified in the RASSMon configuration file.

As the data is stored to files within the filesystem tree of the frontend, the frontend in turn serves pages that use asynchronous HTTP calls (via jQuery / AJAX calls) to query a list of files and deserialize files to JavaScript objects for further processing. Given the standardization of the data formats, RASSMon is able to couple with other software packages such as DataTables [28] for rendering a real-time searchable table or HighCharts [29] for rendering stacked bar charts to track the evolution of service state changes. The project has adopted the stacked-bar charts display for monitoring (Fig. 5) because stacked-bar charts provide an overview of the transitions between service states reported by Check MK. Although the RASSMon CERN plugin is designed to take regular snapshots at fixed time intervals of service states, when the interface is accessed via the frontend, the website will aggregate all captured snapshots into a continuous display. While the backend silently harvests and processes data from various sources, the RASSMon frontend will update the stacked bar chart display in real time accordingly.

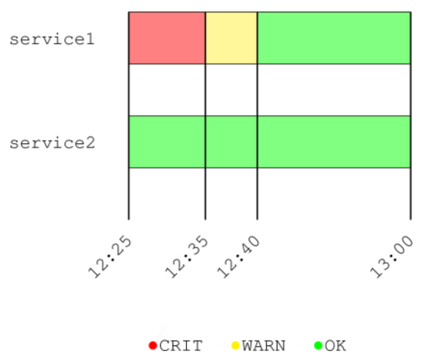

Fig. 5. Stacked bar charts display of services showing the transition between states CRIT, WARN and OK based on snapshots of the service states taken at regular time intervals. RASSMon will merge time intervals during which service states have not been altered.

RASSMon is not limited to a stacked bar charts specifically and due to the HighCharts software package any chart display could be rendered just by modifying the chart type without having to alter code in the RASSMon backend.

VII. Packaging and Distribution

All the software packages used by the RASSMon frontend are installed through the NodeJS package manager NPM in order to separate vendor code from the code written specifically for RASSMon. Code written specifically for RASSMon as well as any overrides are filed separately from vendor code. As an extra added benefit, the third-party packages can be upgraded seamlessly without having to alter the code base.

Using NPM facilitates the deployment of RASSMon by avoiding the inclusion of all dependencies as part of the main distributable software package. Instead, the project can be checked out of the development repository and then all dependencies can be installed via NPM [30].

Considering that RASSMon will be released some time into the future, the developers are considering distributing RASSMon through NPM as a NodeJS package. Other alternatives would include releasing the project through Bower [31] although the latter package management system is geared towards frontend libraries and not self-stand projects. In the previous iterations of RASSMon Bower had been used for frontend components but the current development has adopted NPM instead; primarily due to Bower slowly reaching end of life as stated by the developers [31].

VIII. Conclusions

Even though the Check MK site that RASSMon polls only reports statuses with a considerable delay (up to a half or full hour), it is still useful to report on any status changes as soon as possible. To ensure timely reaction of system administrators, a notifier plugin will be added to send short text messages via an “Short Message Service” (SMS) cell phone gateway.

The project imports Check MK as a NodeJS plugin in order to query the CERN ETF website but other plugins could be written as NodeJS modules to facilitate poll data from different data sources other than Check MK. Based on the RASSMon configuration file and the general code design, it is possible for RASSMon to query other Check MK sites, other than the one provided by CERN for grid monitoring.

RASSMon is in an ongoing development stage but well into the testing phase of the project where the software is being checked for consistency by the system administrators and operators responsible with local grid sites. The feedback gathered contributes towards fixing issues, improving the software and preparing the project for an official public release.

Acknowledgments

The authors would like to thank Dr. Ionut Vasile and Eng. Dragos Ciobanu-Zabet for valuable discussions, and Dr. Mihnea Dulea for continuous support while developing the RASSMon project.

References

[1] C.E.R.N. The worldwide lhc computing grid collaboration (wlcg), http://wlcg.web.cern.ch/.

[2] C.E.R.N. The european organization for nuclear research (CERN), http://cern.ch

[3] A. Duarte, P. Nyczyk, A. Retico, and D. Vicinanza, "Monitoring the egee/wlcg grid services", J. Phys. Conf. Ser., vol. 119, 052014, 2008.

[4] F. Stagni, R. Santinelli, M. Cattaneo, and S. Roiser, "LHCb distributed computing operations," J. Phys. Conf. Ser., vol. 368, 012009, 2012.

[5] M. Kettner, "Check MK development website", http://mathias-kettner.de/check_mk.html.

[6] Ethan Galstad et al., "Nagios: The industry standard in it infrastructure monitoring", https://www.nagios.org.

[7] C.E.R.N. Experiments test framework (ETF), https://etf.cern.ch/.

[8] A. Tsaregorodtsev et al., "Dirac: a community grid solution", J. Phys. Conf. Ser., vol. 119, 062048, 2008.

[9] T. Jim et al., "Cyclone: A safe dialect of C", USENIX Annual Technical Conference, General Track, pp. 275-288, 2002.

[10] E.D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++", ACM sigplan notices, vol. 44, pp. 81-96, 2009.

[11] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad, "Facile: A symmetric integration of concurrent and functional programming," International Journal of Parallel Programming, 18(2):121-160, 1989.

[12] Erich Gamma, "Design patterns: elements of reusable object-oriented software," Pearson Education India, 1995.

[13] Yakov Shafranovich, "Rfc 4180: Common format and mime type for comma-separated values (csv) ?les, 2005," RFC 1654, RFC Editor, 2005.

[14] Stefan Tilkov and Steve Vinoski, "NodeJS: Using javascript to build high-performance network programs," IEEE Internet Computing, 14(6):80-83, 2010.

[15] Mark Otto, Jacob Thornton, Chris Rebert, Julian Thilo, et al., "Twitter bootstrap," Dostupné z: http://twitter.github.io/bootstrap/base-css.html, 2011.

[16] Horia Hulubei National Institute for R&D in Physics and Nuclear Engineering (IFIN-HH). Rassmon: Public demonstration page. https://turing.nipne.ro:3000/.

[17] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[18] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml) version 1.1. yaml. org, Tech. Rep, 2005.

[19] Samudra Gupta, "Logging in Java with the JDK 1.4 Logging API and Apache log4j," Apress, 2003.

[20] Horia V Corcalciuc, "A taxonomy of time and state attacks," in Availability, Reliability and Security (ARES), 2012 Seventh International Conference on, pages 564-573. IEEE, 2012.

[21] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. IEEE communications magazine, 32(9):40–48, 1994.

[22] Clem Baker-Finch, David J King, and Phil Trinder, "An operational semantics for parallel lazy evaluation," in ACM SIGPLAN Notices, volume 35, pages 162-173. ACM, 2000.

[23] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in javascript," in Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on, pages 1-10. IEEE, 2015.

[24] KAHN Gilles, "The semantics of a simple language for parallel programming," in Information Processing, 74:471-475, 1974.

[25] Gilles Kahn and David MacQueen, "Coroutines and networks of parallel processes," 1976.

[26] Kai Lei, Yining Ma, and Zhi Tan, "Performance comparison and evaluation of web development technologies in php, python, and node. js," in Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on, pages 661-668. IEEE, 2014.

[27] Daniela Florescu, Alon Y Levy, and Alberto O Mendelzon. Database techniques for the world-wide web: A survey. SIGMOD record, 27(3):59–74, 1998.

[28] SpryMedia Ltd, "Datatables: Table plug-in for jquery," 2007.

[29] AS Highsoft, "Highcharts-interactive javascript charts for your web projects," 2014, 2014.

[30] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean, "When github meets cran: An analysis of inter-repository package dependency problems," in Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, volume 1, pages 493-504. IEEE, 2016.

[31] Twitter, "Bower - a package manager for the web," https://github.com/bower/bower, 2016.