Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

NODE.JS

ESTUDO TECNOLÓGICO E DESENVOLVIMENTO FULL-STACK JAVASCRIPT DE PLATAFORMA DE COMPETIÇÕES EM PROBLEMAS ALGORÍTMICOS

TRABALHO DE GRADUAÇÃO

GUSTAVO STOR DE AGUIAR

Recife, Julho de 2015

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

NODE.JS

ESTUDO TECNOLÓGICO E DESENVOLVIMENTO FULL-STACK JAVASCRIPT DE PLATAFORMA DE COMPETIÇÕES EM PROBLEMAS ALGORÍTMICOS

ALUNO: **GUSTAVO STOR DE AGUIAR**
ORIENTADOR: **FERNANDO DA FONSECA DE SOUZA**

Trabalho de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Recife, Julho de 2015

Agradecimentos

Foram quatro anos e meio que cursei Ciência da Computação no Centro de Informática da Universidade Federal de Pernambuco. Nesse tempo, não só conheci amigos maravilhosos, como grupos nos quais me identifiquei bastante. O PET-Informática, que me acolheu de Setembro de 2011 a Fevereiro de 2015, foi extremamente importante para o meu desenvolvimento ético, filantrópico e de trabalho em equipe. No PET, fiz amizades que perduram até hoje. Muito obrigado a todos que trilharam alguma parte da história comigo nesse grupo fenomenal.

Participar da Maratona de Programação foi o meu maior pilar de força de vontade e coragem. Fazer parte dessa rotina, todas as manhãs e tardes de sábado, foi uma das decisões mais certas que eu já fiz. Foi com a Maratona de Programação que eu transformei a arte de programar em uma paixão. A maior parte das minhas conquistas fora do meio acadêmico, como ter recebido uma proposta de trabalho do Facebook, Google e Microsoft, foram devido ao conhecimento que obtive nesse grupo. Agradeço a todos que trilharam essa rotina comigo, em particular à professora Liliane Salgado, que leva esse projeto com tanta garra e vigor, e a todos que participaram de algum time comigo: Bruno, Sabóia, Lucas Veras, Rael, Lucas Lima, Mário e Duhan.

Agradeço aos professores Fernando da Fonseca de Souza, mentor brilhante, tutor do PET, excelente professor e de quem eu já tive oportunidade de ser monitor por alguns semestres; e Kátia Guimarães, professora de algoritmos e estruturas de dados, do qual fui monitor por dois anos, brilhante academicamente e pessoalmente, com seu carisma e vontade de ensinar. A todos os demais professores, agradeço por vocês se esforçarem para alavancar o reconhecimento do CIn mundo afora.

Por fim, agradeço à minha família. À Marlene, uma segunda mãe para mim. A meus irmãos, Camila e Guilherme, que sempre estão ao meu lado torcendo por mim. A meus cunhados, Diego e Thais, que também considero irmãos, por tanto tempo que nos conhecemos, desejo-lhes o melhor que a vida possa oferecer. A meus pais, Antônio e Marta, eu só queria que vocês soubessem que eu me considero extremamente sortudo de ter vocês como pais, por me amarem acima de qualquer coisa, me apoiarem em todas as minhas decisões e me aceitarem por quem sou: um jovem cientista da computação com grandes ambições. Eu sou extremamente grato a vocês. E a Victor, a quem eu agradeço por estar sempre perto, e compartilhar dos mesmos sonhos. Obrigado por fazer meus dias mais felizes.

"*Science is a differential equation. Religion is a boundary condition.”*
(Alan Turing)

RESUMO

Desenvolvimento de software para a Web é uma das áreas mais requisitadas no mercado de tecnologia da informação. As decisões envolvidas com o projeto de uma aplicação online variam desde a escolha do sistema operacional até qual a tecnologia de desenvolvimento que será utilizada. No meio da crescente demanda em plataformas de desenvolvimento Web de alto desempenho, e que facilitem a criação de aplicações dinâmicas na Web, Node.js surge não só para solucionar, mas também para mudar o paradigma de desenvolvimento Web a que estamos acostumados — integrando a mesma linguagem de programação no cliente e no servidor. Os principais objetivos desse trabalho envolvem investigar detalhadamente o funcionamento dessa plataforma e sua arquitetura, realizar diversos testes comparativos entre essa e outras tecnologias e implementar uma aplicação em Node.js.

A aplicação desenvolvida neste trabalho é uma plataforma de competições simuladas de programação, a fim de prover uma sistematização dos treinos em programação competitiva nas instituições de ensino brasileiras. Muitos times que representam o Brasil em competições mundiais de programação, como a Olimpíada Internacional de Informática (IOI) e a Competição Internacional de Programação para o Colegiado (ICPC) não possuem nenhum método de treino, resolvendo problemas algorítmicos avulsos para ganhar experiência e conhecimento. Na aplicação desenvolvida, qualquer instituição ou aluno individualmente pode criar suas próprias competições virtuais, nas mesmas regras das competições oficiais, e escolher as questões a partir de vários repositórios de problemas algorítmicos na Internet. Com esse sistema, não somente colocamos em prática o conhecimento de Node.js elucidado neste trabalho, como também plantamos o sonho de ampliar a visibilidade do potencial do Brasil mundo afora.

Palavras-chave: desenvolvimento Web; Node.js; programação competitiva.

ABSTRACT

Software development for the Web is one of the most sought after areas in the information technology market. Decisions involved with the design of an online application range from the choice of the operating system to which development technology will be used. Amid the growing demand for high-performance Web development platforms and which facilitate the creation of dynamic Web applications, Node.js comes not only to solve these issues but also to change the Web development paradigm to which we are used to — by integrating the same programming language on the client and server. The main goals of this work involves a detailed investigation of the operation of this platform and its architecture, making various comparative tests between this and other technologies and deploying an application in Node.js.

The application developed in this work is a platform for simulated programming competitions simulated, in order to provide a systematisation of training for programming contests in Brazilian educational institutions. Many teams representing Brazil in global programming competitions such as the International Olympiad in Informatics (IOI) and the International Collegiate Programming Contest (ICPC) have no training method, but perform loose algorithmic problem solving to gain experience and knowledge. In the developed application, any institution or individual student can create their own virtual competitions under the same rules of the official competitions, and choose the questions from various repositories of algorithmic problems on the Internet. With this system, we are not only putting into practice the knowledge of Node.js elucidated in this work, but also planting the dream of increasing the visibility of Brazil's potential around the world.

Keywords: Web development; Node.js; competitive programming.

SUMÁRIO

INTRODUÇÃO

A World Wide Web é, de longe, o serviço mais utilizado da Internet. Por meio dela, conseguimos obter conteúdo e nos conectar a pessoas em poucos milissegundos. Todo esse conteúdo e essas pessoas a quem nos conectamos estão disponíveis em páginas na Web, e a ordem de grandeza dessas páginas existentes é a mesma que a quantidade de pessoas no mundo[1]. É como se cada pessoa do mundo fosse responsável por uma página virtual. A importância da World Wide Web nos dias de hoje é de tamanha magnitude que o desenvolvimento de aplicações Web passou a ser um dos conhecimentos mais requisitados no mercado de trabalho em tecnologia de informação.

Mas tudo é feito de escolhas, e nem sempre são fáceis. Desenvolver uma aplicação na Web começa sempre com uma escolha: que tecnologia usar? Um iniciante poderia escolher a que provê um conjunto de ferramentas ou frameworks que facilite o desenvolvimento; um desenvolvedor mais experiente poderia optar por uma que ofereça suporte a mais conexões paralelas, ou que seja otimizada no gerenciamento de requisições. Existem inúmeras linguagens de programação com suporte a desenvolvimento Web, algumas com seus respectivos frameworks que auxiliam no desenvolvimento: Django[2], em Python[3]; Rails[4], em Ruby[5]; PHP[6]. A escolha da tecnologia utilizada depende muito do desenvolvedor, dos objetivos e da determinação do desenvolvedor.

Dizer qual tecnologia de desenvolvimento Web é mais apropriada para um determinado projeto depende muito dos requisitos. Não muito longe disso, porém, é comum querer comparar tecnologias em determinados aspectos específicos. “Qual é a melhor tecnologia a utilizar quando queremos otimizar o tempo de resposta de uma requisição?”, “Qual é a tecnologia mais propícia para iniciantes?”, “Que tecnologia podemos usar para desenvolver um servidor Web que facilite a persistência de conexão cliente-servidor?” ou “É possível uma única instância de servidor aguentar ao menos 10

[1] Dado extraído de http://www.worldwidewebsize.com/ em Abril de 2015.
[2] https://www.djangoproject.com
[3] https://www.python.org
[4] http://rubyonrails.org
[5] http://rubyonrails.org
[6] http://php.net

mil conexões paralelas?" são perguntas comuns e que podemos, neste caso, responder precisamente qual ou quais tecnologias disponíveis são mais apropriadas para o determinado requisito.

O último ponto levantado – uma implementação de servidor Web que consiga manter 10 mil conexões paralelas – é frequentemente revisitado em pesquisas e levantamentos das tecnologias que melhor atendem às demandas de grandes servidores em produção. O problema C10K[7] relata justamente isso: o gargalo não é mais o hardware, pois com uma máquina 2000MHz, com 4GB de memória RAM e uma placa de rede de 1000Mbits/s, dividindo esses recursos para 10000 clientes, disponibilizaríamos, em um simples cálculo baseado na distribuição uniforme dos recursos, 200KHz de processamento, 400KB de memória virtual e uma conexão de 100Kbits/s para cada usuário, o que pode ser bastante suficiente para a maioria das páginas na Web. Mas o mundo não é tão ideal assim, e nem tudo depende do hardware. Chegamos em um ponto em que o problema maior é desenvolver o software que consiga ser o mais eficiente possível na distribuição dos recursos. Numa era em que 5 segundos de espera é muito tempo, otimização dos recursos pode ser a melhor arma para quem quer se sobressair.

O problema que mencionamos foca no lado do cliente, mas desenvolvimento Web também deve se importar com o lado do desenvolvedor. Nesse âmbito, existem práticas que podem facilitar muito o desenvolvimento. Primeiramente, a tecnologia escolhida deve dispor de um *framework* de desenvolvimento que seja continuamente mantido por uma equipe ou comunidade de desenvolvedores – dependendo de sua natureza privada ou pública. Esse aspecto é fundamental em computação, pois códigos se tornam obsoletos muito rapidamente. Erros que comprometem a segurança são encontrados a todo instante e as tecnologias devem acompanhar a passo rápido essas mudanças e correções. Outro fundamental pré-requisito é que a tecnologia empregada disponha de técnicas atuais de programação. Desenvolver um servidor Web em Perl[8], como era comum no fim da década de 90, seria considerado um pesadelo nos dias de hoje, com tantas opções mais flexíveis existentes. Por fim, muitos defendem a ideia de que o desenvolvimento se torna mais prático e modular quando utilizamos a mesma linguagem de programação para desenvolver o *frontend* e o *backend* do servidor Web.

[7] http://www.kegel.com/c10k.html/
[8] https://www.perl.org

Tendo em vista todos os problemas levantados, uma nova tecnologia foi proposta e colocada em prática há poucos anos, e a comunidade de desenvolvedores dela está cada vez maior. Focada em desenvolvimento de servidores Web em JavaScript[9], a plataforma Node.js[10] foi implementada utilizando o mecanismo V8, desenvolvido pelo Google[11]. V8 é um mecanismo altamente otimizado de JavaScript que funciona como um compilador JIT (Just-In-Time) — conceito introduzido por McCarthy (1960) — e transforma o código alto-nível em código de máquina. Assim como o Google Chrome[12], que também utiliza o mecanismo V8, o Node.js incorporou esse mecanismo e desenvolveu uma plataforma para desenvolvimento Web, que trás consigo os benefícios de JavaScript no lado do servidor, e os benefícios do desempenho do V8 como base estrutural da tecnologia.

Node.js é principalmente utilizado em sistemas com excesso de operações de entrada e saída, de forma que essas operações não bloqueiam o processador do servidor e executam em paralelo no sistema operacional. Além disso, a aplicação principal é executada em uma única thread, abstraindo para o programador a criação de outras threads em decorrência de operações não-bloqueantes. Para entender melhor como a tecnologia consegue suportar milhares de conexões e manter sua alta eficiência, fazemos um estudo detalhado nesse trabalho sobre Node.js e as técnicas de otimização exercidas no interior da plataforma, além de vários testes comparativos com outras linguagens de programação e tecnologias de desenvolvimento Web. Os testes feitos no decorrer do trabalho comparam a eficiência do JavaScript V8 em termos de velocidade de processamento, linhas de código para uma determinada tarefa, e recursos do sistema (memória e CPU), com outras linguagens de programação. Comparam também o desempenho do Node.js, que tem o V8 como um dos seus principais componentes, em termos de quantidade de requisições que o servidor consegue lidar por segundo.

Além do estudo tecnológico de Node.js, este trabalho envolve a concepção e implementação de uma aplicação utilizando essa tecnologia. A aplicação desenvolvida tem por objetivo prover um meio de sistematizar treinos de programação competitiva no Brasil. Durante o desenvolvimento, colocamos em prática todo o conhecimento da tecnologia Node.js abordado neste trabalho, de forma a maximizar o desempenho de nossa aplicação e prover o mínimo de latência possível para as requisições dos usuários.

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[10] https://nodejs.org/
[11] http://code.google.com/apis/v8/
[12] https://www.google.com/chrome/

Estabelecemos metas e aspectos funcionais e não-funcionais que foram implementados para que a aplicação cumpra seu papel. Os aspectos funcionais envolvem as características do sistema no nível de comportamento e funções, enquanto os não-funcionais definem como o sistema é operado, considerando critérios de desempenho, usabilidade, tecnologias utilizadas e outros.

O primeiro capítulo deste trabalho relembra alguns conceitos básicos da Web, abordando rapidamente a evolução do desenvolvimento Web desde que o primeiro protocolo HTTP (BERNERS-LEE, 1991) foi estabelecido, na década de 90. Falamos também sobre a Web interativa e sobre como o JavaScript rapidamente dominou a linguagem de script em todos os navegadores, mesmo sendo tão popularmente criticada. Das duas últimas seções do capítulo, uma discorre brevemente sobre como funciona o fluxo de dados na Web, seguindo a pilha de protocolos TCP/IP; e outra explica como funciona o fluxo de requisições no servidor Web, isto é, o que acontece depois que a requisição viaja pela pilha de protocolos TCP/IP até chegar no servidor.

No Capítulo 2, estudamos o que é o V8 e como ele torna o JavaScript tão rápido. Entendemos as otimizações feitas por este mecanismo e ainda realizamos várias avaliações de desempenho com outras tecnologias, para realmente medir se esse mecanismo torna a execução de código JavaScript mais rápida que outras linguagens de programação. Logo adiante, no terceiro capítulo, fazemos um estudo tecnológico da plataforma Node.js propriamente dita. Começamos abordando o contexto histórico e como o Node.js surgiu, e seguimos com as características e arquitetura da tecnologia que a tornam única em seus benefícios. Por fim, utilizamos uma série de *benchmarks* para comparar a média de requisições por segundo e tempo médio por requisição da plataforma, em critérios como servir arquivos estáticos, operações de I/O e operações de CPU.

O capítulo seguinte aborda a aplicação desenvolvida, no qual falamos sobre a problemática e o nosso objetivo com o projeto. Explicamos também cada aspecto funcional e não-funcional implementado, de forma que focamos o máximo possível em tornar a interação do usuário com o sistema uma experiência fluida e rápida. Terminamos o capítulo quatro com a arquitetura da aplicação e os resultados dos testes que realizamos com usuários reais. Por fim, no último capítulo, discutimos sobre a visão geral deste trabalho e as conclusões que tiramos com a tecnologia estudada, assim como as mudanças que almejamos implementar no futuro.

1 A WEB

Antes de estudarmos o Node.js e implementarmos uma aplicação utilizando essa plataforma, precisamos entender e relembrar alguns conceitos importantes da Web. Começamos este capítulo falando um pouco sobre como se deu a evolução do desenvolvimento Web, desde a concepção do seu primeiro protocolo até o surgimento das primeiras páginas interativas. Terminaremos este capítulo explicando brevemente como funciona o fluxo de comunicação na Internet, para poder então discutir os passos que cada requisição segue desde que entra no servidor até gerar uma resposta ao cliente; compararemos dois servidores Web bastante utilizados no mercado de tecnologia, e um deles será utilizado em conjunto com nossa aplicação para alavancar ainda mais o desempenho da plataforma desenvolvida neste trabalho.

1.1 EVOLUÇÃO DO DESENVOLVIMENTO WEB

Há 60 mil anos, construir uma casa não era tão simples quanto nos dias de hoje. O mundo não era conectado. As informações e técnicas que uma comunidade desenvolvia para as construções não eram amplamente conhecidas. As pessoas tinham que passar pelo mesmo ciclo de descoberta e aprendizado para construir suas próprias ferramentas. Com o passar do tempo, as informações passaram a circular, o homem passou a vender seus serviços e logo já havia indústrias especializadas em construções. Algumas indústrias focavam na fabricação de furadeiras, outras na de tijolos, mas o propósito era comum: auxiliar na construção de uma casa.

Com a Web não foi muito diferente. No início dos anos 90, quando o protocolo HTTP foi introduzido, a Internet era muito estática. Quando uma pessoa requisitava o conteúdo de uma página *online*, ela só recebia texto. Suas opções como desenvolvedor eram um tanto limitada. Os desenvolvedores passavam pelo mesmo ciclo de desenvolvimento na concepção de uma aplicação na Internet, e muito pouco código era reaproveitado. Não existiam f*rameworks*. Mas a comunidade de programadores ansiava por mais, e era claro o horizonte de opções que um simples

HTML estático abriu. A casa a que nos referimos aqui era o conteúdo que gostaríamos de disponibilizar *online*, e as ferramentas viriam a ser uma série de opções de desenvolvimento que surgiriam nos anos seguintes.

No começo da Web, os servidores que processavam as requisições HTTP eram quase todos implementados em C[13] e serviam páginas estáticas correspondentes ao diretório extraído da URL[14]. Se buscássemos o endereço www.foo.com/bar.html, o servidor seria responsável por retornar o arquivo estático bar.html. Em 1993, a *Common Gateway Interface* (CGI) [15] foi introduzida pelo Centro Nacional de Aplicações em Supercomputação (NCSA)[16], um órgão estadounidense focado em pesquisas em informática e telecomunicações. Agora, a funcionalidade padrão de um servidor Web poderia ser estendida para que o diretório apontado pela URL não se limitasse apenas ao HTML estático, mas também pudesse ser um script executável. Se este fosse o caso, ao invés de retornar o arquivo, este seria executado e a resposta gerada (geralmente HTML) seria retornada ao servidor para, por fim, gerar uma resposta para o cliente. E assim surgiram as primeiras páginas com conteúdo gerado dinamicamente na Web.

Uma série de *frameworks* para desenvolvimento Web começaram a surgir nos anos seguintes, em diversas linguagens como, por exemplo, Python, PHP e Java[17]. Alguns surgiram para suprir a necessidade de grandes corporações, que visionavam grandes oportunidades de negócio nessa nova era virtual. Alguns outros surgiram com comunidades *open source* de programadores que simplesmente buscavam resolver seus problemas. Esses *frameworks* passaram a abstrair complexidades subjacentes da Web e códigos *boilerplate*[18] que todo desenvolvedor deveria escrever para ter um servidor funcionando. Eles proveram uma inversão de controle nesse âmbito de desenvolvimento, no qual o desenvolvedor só deveria se preocupar com o aspecto lógico de sua aplicação, e não mais com os protocolos de comunicação.

---

[13] http://www.cprogramming.com/
[14] *Uniform Resource Locator* é o padrão de endereço global para documentos e outros recursos na Web.
[15] http://www.w3.org/CGI/
[16] http://www.ncsa.illinois.edu/
[17] https://www.java.com/
[18] Seção de código que deve ser incluída em múltiplos lugares com nenhuma ou quase nenhuma alteração.

Em 2000, apenas sete anos após a primeira página dinâmica, tecnologias como PHP, ASP[19] e Java já dominavam a maior parte de servidores Web existentes. Uma década viria a se passar e muitos *frameworks* ainda iriam surgir até o lançamento da primeira versão do Node.js. No entanto, para o Node.js ser concebido, o JavaScript precisava nascer antes.

1.2 JAVASCRIPT: WEB INTERATIVA

Em 1995, numa tentativa de fazer o recém adicionado suporte à Java applets[20] no navegador da Netscape[21] acessível para Web *designers* e programadores não experientes em Java, Brendan Eich, então recentemente contratado pela empresa, desenvolveu uma linguagem de script poderosa. Chamada inicialmente de Mocha, e posteriormente de LiveScript, foi nomeada de Javascript após a Sun[22] torná-la uma marca registrada. Em pouco tempo, JavaScript começou a expandir seu propósito, que inicialmente era apenas manipular Java applets. Ela tornou o acesso aos elementos de HTML, como formulários, botões e imagens muito mais fácil para os desenvolvedores Web. A linguagem de programação começou a ser usada para manipular imagens e o conteúdo das páginas e, ao fim da década de 90, já havia *scripts* que simplesmente mudavam uma imagem para outra em resposta a eventos gerados pela movimentação do *mouse*.

JavaScript é uma linguagem interpretada, isto é, seu código é lido e interpretado por outro programa, chamado de interpretador (no nosso caso, o próprio navegador), e normalmente não é transformada em código de máquina[23]. Assim sendo, ela não requer um compilador, o que atraía bastante entusiastas da filosofia de que "*o quanto mais simples as coisas são no lado do cliente, melhor*". Isso fez com que muitos a considerassem apenas um brinquedo, ridicularizassem e afirmassem que não se

[19] http://www.asp.net/
[20] Tipo especial de programa Java que um navegador habilitado com a tecnologia Java pode baixar da internet e executar.
[21] Empresa de serviços de computadores dos EUA, mais conhecida pelo seu navegador Web.
[22] http://www.oracle.com/us/sun/
[23] O Google Chrome foi implementado em cima do V8, um mecanismo de JavaScript que compila o script para código de máquina afim de potencializar a eficiência do mesmo.

precisava de habilidade alguma para se programar JavaScript. Eventualmente, uma sombra enorme de menosprezo cobriu o extraordinário potencial que esta linguagem de programação tinha a oferecer para a Web. Muitos anos depois, quando o Node.js surgiu e corrigiu uma parte dos “lados ruins” de JavaScript, uma grande comunidade de desenvolvedores adotou-a em seus servidores web e passou a considerá-la uma das opções mais viáveis e escaláveis para desenvolvimento de projetos, mudando a forma como muitos programadores pensavam sobre a linguagem.

Javascript possuía uma série de falhas de segurança em seus primórdios. Ter sido desenvolvida em apenas dez dias pode ter contribuído para isso, mas isso não impediu de, em pouco tempo, a Associação Européia de Fabricantes de Computadores (ECMA)[24] criar uma especificação padrão para ela, de forma que desenvolvedores de outros navegadores pudessem implementar um interpretador. O padrão ficou conhecido como ECMAScript (1999). Isso foi necessário principalmente devido a falta de compatibilidade de código *frontend* entre múltiplos navegadores, e mediante à adoção em massa de Javascript por desenvolvedores Web, o que apressava as grandes empresas a implementar um mecanismo compatível com essa linguagem em seus navegadores.

Um fato bastante disseminado no meio tecnológico é que o JavaScript teve muito pouco tempo para ser produzido, e por conseguinte nasceu com inúmeros erros e más práticas. Segundo Crockford (2008, p. 1, tradução nossa):

> JavaScript é uma linguagem com mais partes ruins do que o normal. Foi de uma não-existência à adoção global em um alarmante pequeno período de tempo. Ela nunca passou por um período de teste em que pudesse ser experimentada e polida. Foi direto ao Netscape Navigator 2 do jeito que estava, e estava muito rudimentar. Quando os Java applets falharam, JavaScript se tornou a “Linguagem da Web” por padrão. A popularidade do JavaScript é quase completamente independente de suas qualidades como linguagem de programação.

Esses erros não podiam ser consertados pelo comitê de padronização, pois podiam causar uma falha em série em todos os scripts na Web que utilizavam algo de um jeito não recomendado. O melhor que os comitês poderiam fazer é lançar alguma mudança que amenizasse os sintomas de erros cometidos em versões antigas, ou introduzisse novas formas de se produzir uma mesma tarefa; em ambos os casos,

[24] http://www.ecma-international.org/

muitas vezes essas mudanças acabavam por não se dar bem com as partes ruins da linguagem, o que eventualmente gerava mais partes ruins.

Mas nem tudo são coisas ruins. O próprio Crockford, arquiteto sênior de JavaScript no PayPal[25], conhecido por introduzir o JSON (*JavaScript Object Notation*) (2006) membro do comitê de ECMAScript, afirma que "toda linguagem tem sua parcela de partes ruins, mas somos nós responsáveis por usá-las ou não […]. As partes boas de JavaScript estão tão escondidas que por muitos anos a linguagem foi considerada um brinquedo sem graça e incompetente" (2008, p.2-3).

Aprender a programar bem em JavaScript, ao invés de dedicar o mesmo tempo aprendendo outras linguagens não tão criticadas no mercado, tem três grandes razões: a primeira, e mais óbvia, é que se o objetivo for desenvolver para *frontend* Web (a camada visível ou que interage de perto com o usuário), esta é a única possibilidade. A Web se tornou uma grande plataforma de desenvolvimento, e JavaScript se tornou o único padrão encontrado em todos os navegadores. Segundo, porque mesmo para um desenvolvedor de *backend*, Node.js é uma excelente opção a se considerar na hora de começar o projeto, como veremos mais à frente. Node.js não segue à risca o padrão ECMAScript — até porque ele não foi feito para navegadores — e eliminou alguns dos piores aspectos da linguagem, como a natureza global das variáveis. Terceiro, que independente do tipo de desenvolvedor, JavaScript na verdade é uma excelente linguagem quando quem a utiliza, realmente a domina. É leve, expressiva e funcional, e segue o modelo orientado a eventos que é ideal tanto para o servidor, quanto para o cliente.

1.3 FLUXO DE COMUNICAÇÃO NA WEB

Quando visitamos um determinado endereço na Web, a requisição segue uma série de passos e protocolos até se transformar em uma resposta visível no navegador. Não estamos interessados neste trabalho em estudar a fundo o fluxo de comunicação na Web, que vai desde o encapsulamento de sua requisição em uma

[25] https://www.paypal.com

mensagem HTTP até a divisão dos dados em meio físico; estamos interessados em entender os passos que a requisição segue desde o momento em que chega na aplicação do servidor. Mas antes disso, precisamos relembrar alguns conceitos básicos sobre a Internet.

Figura 1: Pilha de protocolos TCP/IP.

Fonte: O Autor

Quem entende um pouco sobre a infraestrutura da Internet, deve conhecer a pilha de protocolos TCP/IP (Figura 1). Essa pilha define as diversas etapas que uma requisição passa para chegar da aplicação cliente (no nosso caso, o navegador) até a aplicação destino (o servidor web). Estamos acostumados a definir as camadas da seguinte maneira:

Aplicação - processos nessa camada são específicos da aplicação, e os dados são codificados por algum protocolo padrão como HTTP, FTP (POSTEL, 1985), SMTP (POSTEL, 1982), entre outros. O pacote nessa camada é chamado de mensagem.

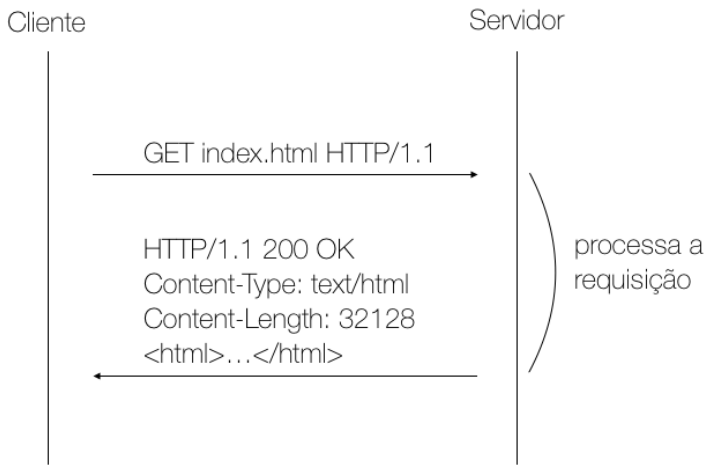
Transporte - camada responsável por determinar se as mensagens sempre chegam (confiabilidade) e se chegam na ordem correta (integridade). Distingue as aplicações subjacentes por um número de porta e utiliza algum protocolo de transporte como TCP (POSTEL, 1981) ou UDP (POSTEL, 1980). O pacote nessa camada é chamado de segmento.

Rede - camada responsável por transmitir os segmentos entre pontos físicos distintos, para fazer os dados chegarem da fonte ao destino, por meio de um processo denominado roteamento. Distingue os sistemas utilizando o endereço IP. É o principal protocolo na pilha de protocolos TCP/IP. O pacote nessa camada é chamado de datagrama.

Física - é a camada de nível mais baixo na pilha, e é dependente do sistema físico no qual opera. Trata das tecnologias de comunicação em meio físico, com fio ou sem fio. É responsável também pela tradução de endereços lógicos em endereços físicos.

Uma requisição segue essa pilha de protocolos para conseguir chegar da fonte ao destino. O protocolo mais perto da aplicação está, obviamente, na camada de aplicação; FTP é o protocolo utilizado por aplicações de transferências de arquivos entre computadores da rede; SMTP é o protocolo utilizado por algumas aplicações de email; HTTP é o protocolo usado no nosso escopo de estudo, pois é utilizado para aplicações no modelo cliente-servidor para requisição e resposta de conteúdo da Web. HTTP é o protocolo de aplicação mais conhecido da Internet, pois é ele o responsável por permitir que acessemos páginas *online*.

Figura 2: Diagrama de sequência de uma simples requisição GET.



Fonte: O Autor

O modelo simplificado da Figura 2 mostra uma requisição HTTP. Nesse protocolo, há alguns métodos diferentes de requisição. Os mais conhecidos são GET e POST (BERNERS-LEE at al., 1997). O objetivo do GET é simplesmente recuperar dados, e não deve ter efeito colateral no cliente. Na maior parte das vezes, o conteúdo que se deseja reaver está definido no próprio endereço URL da requisição, muitas vezes junto com *query strings*, que é uma forma de se passar parâmetros na requisição. Por exemplo, no endereço fictício http://www.foo.com/r/bar.html?year=2015, a primeira parte (www.foo.com) forma o domínio, e serve para identificar o hóspede do servidor na Internet; a segunda parte

(/r/bar.html) forma o caminho do conteúdo dentro do diretório do servidor; e a última parte (?year=2015) é a *query string*, e funciona como um refinamento da requisição. Se esse endereço retornasse uma série de notícias marcadas com uma data, esse refinamento poderia servir para retornar apenas as notícias de 2015. Outras vezes, porém, estes parâmetros podem estar escondidos no corpo da mensagem; esta abordagem é utilizada principalmente em websites que possuem sistemas de autenticação, nos quais determinados recursos são acessíveis apenas para usuários específicos.

O método POST é utilizado principalmente quando queremos que alguma ação de armazenamento ou modificação seja feita no servidor. Porém, esse não é seu único uso. A RFC 2616, que especifica o protocolo HTTP 1.1, define que "o método POST é utilizado para requisitar ao servidor destino que este aceite a entidade anexada na requisição como um novo subordinado do recurso identificado pela Request-URI …" (BERNERS-LEE et al., 1997, tradução nossa). Em outras palavras, o conteúdo do POST, que pode ser uma anotação de recursos existentes, uma postagem a um fórum online, um bloco de dados contendo a resposta de um formulário ou qualquer outro dado que se deseje transmitir ao servidor, é utilizado como um dos recursos da requisição, e o servidor terá livre-acesso a ele para fazer qualquer tipo de processamento — que pode ser uma simples autenticação de usuário, sem efeitos colaterais no servidor, ou uma operação mais complexa, como modificar o banco de dados da aplicação.

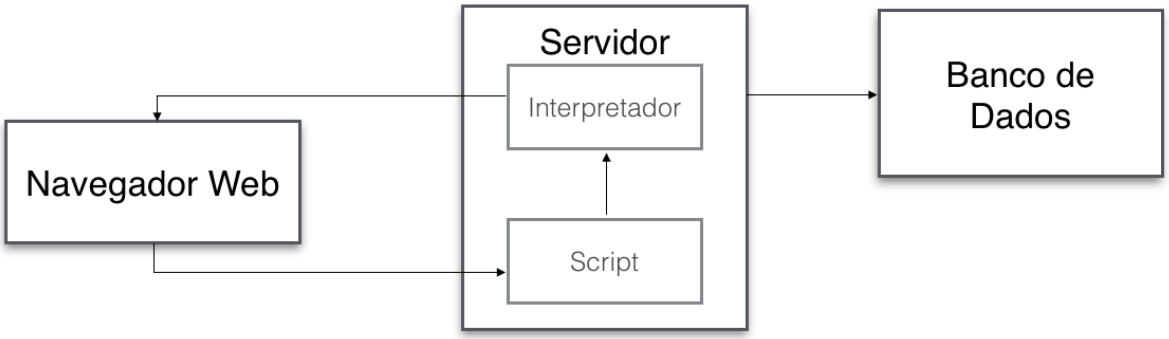
Relembramos até aqui alguns conceitos básicos da Internet, e focamos no protocolo que estamos mais interessados: HTTP. A partir de agora, precisamos entender o que um servidor faz com uma requisição HTTP, dado que esta chega no servidor. Com isso, conseguiremos entender as diferentes abordagens utilizadas no desenvolvimento Web para, então, introduzir a que estudaremos neste trabalho.

1.4 FLUXO DE REQUISIÇÃO NOS SERVIDORES

Assim como a rede mundial de computadores opera sobre uma pilha de protocolos, os servidores web também definem sua própria pilha de etapas. Esta pilha é conhecida como *Web stack* (pilha da Web), ou *HTTP service stack* (pilha de serviços

HTTP). Uma das pilhas de software mais conhecidas é LAMP, acrônimo para Linux[26], Apache[27], MySQL[28] e alguma linguagem de script dentre PHP, Perl e Python. Há outra variante, denominada LEMP, que ao invés de utilizar o servidor Apache, usa o Nginx[29] (pronuncia-se *engine-x*). No fluxo que estamos interessados, podemos abstrair qual o sistema operacional utilizado e qual o modelo de banco de dados da aplicação. O que precisamos notar aqui são dois fatos importantes: o servidor Web e a linguagem de programação. Veremos, no Capítulo 3, que o Node.js por si só já é um servidor Web, mas que podemos integrá-lo com servidores Web mais robustos, como os dois citados, para aumentar ainda mais o poder da plataforma.

Figura 3: Fluxograma simplificado de requisição entre o navegador e o servidor.



Fonte: O Autor

A Figura 3 ilustra simplificadamente o fluxo de uma requisição dentro do servidor utilizando o modelo LAMP. Dizemos que este fluxo é simplificado pois ignora que o servidor muito provavelmente terá um sistema de *cache*, servirá com arquivos estáticos independente ao interpretador subjacente e também poderá ser intermediado por um sistema de balanceamento de cargas; uma grande aplicação Web muito provavelmente terá todas essas características, mas elas são opcionais para sistemas menores.

Neste modelo, o navegador faz uma requisição ao servidor. O recurso solicitado pode ser um arquivo estático, como uma página HTML, ou um arquivo de script. No segundo caso, o servidor é responsável por chamar o módulo interpretador. Este módulo pode fazer alterações ou simplesmente recuperar dados do banco de dados.

[26] Sistema operacional *open source* baseado em Unix.
[27] http://www.apache.org
[28] https://www.mysql.com
[29] http://nginx.org

No final, ele gera uma resposta, que pode ser um arquivo HTML, uma imagem, ou qualquer outra coisa que você considere que um servidor possa retornar. Esta resposta então é devolvida ao servidor Web, que se responsabiliza por encapsulá-la em uma mensagem HTTP e devolver ao cliente.

Com isso, percebemos que o servidor é responsável pela parte burocrática do fluxo: lidar com o recebimento e o envio de mensagens, servir arquivos estáticos, lidar com a cache, entre outros. É como se o servidor Web fosse o recepcionista de uma empresa. Nesse caso, os códigos desenvolvidos são os escritórios. Eles que dizem o que, quando e como a empresa faz o que faz. Eles dirigem o servidor e controlam toda a lógica da aplicação. Mas eles precisam de um segurança (isto é, o servidor Web) para deixar entrar no prédio e guiar ao escritório correto.

Tanto o Nginx quanto o Apache são ótimos servidores Web, e estão no mercado há muitos anos. O Apache foi criado em 1995 por Robert McCool, um desenvolvedor de software que na época trabalhava na NCSA, e também ajudou a construir o CGI, que discutimos mais cedo neste capítulo. O nome original do servidor era *NCSA HTTPd web server*, mas foi posteriormente nomeado de Apache HTTP Server. É o servidor Web mais popular na Internet desde 1996, o que lhe dá a vantagem de ser profundamente documentado e com suporte à integração de diversos outros módulos. É extensível por meio de um sistema de módulos dinamicamente carregados e pode processar linguagens interpretadas, como PHP, Perl e Python. Além disso, o Apache provê módulos de multi-processamento (MPMs), como o mpm_prefork, mpm_worker e mpm_event, que ditam como cada requisição será processada em termos de recursos dedicados a ela[30].

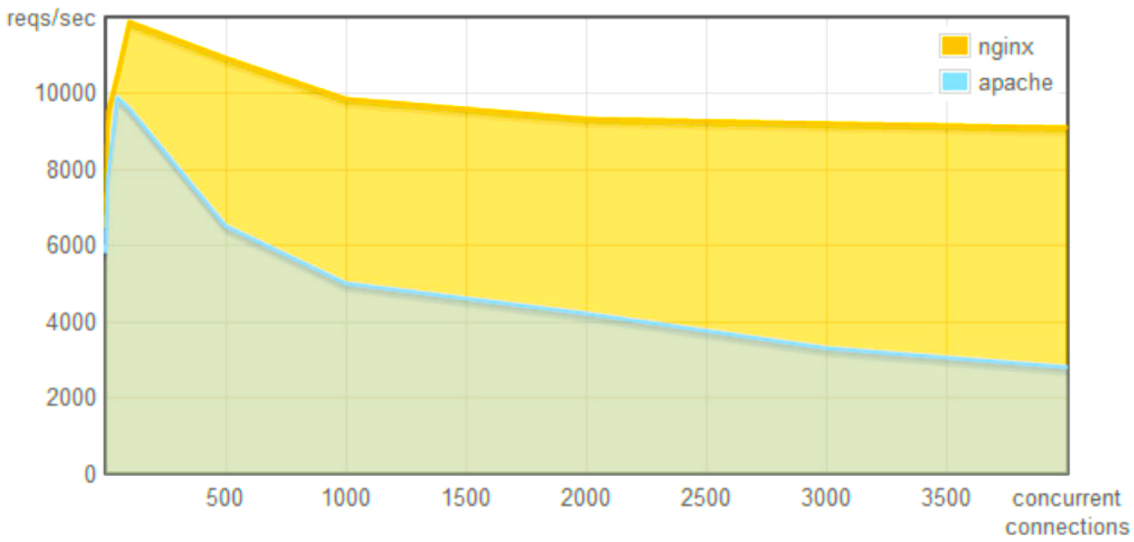
O Nginx é apenas um pouco mais recente. Foi introduzido em 2002 por Igor Sysoev numa tentativa de resolver o problema C10K, que discutimos na introdução deste trabalho. Foi lançado ao público em 2004, quase uma década depois do Apache. Nginx não utiliza muitos recursos e consegue escalar bastante utilizando pouco hardware. Ele é especialista em servir arquivos estáticos de forma muito rápida, e é integrado com outros servidores para lidar com arquivos dinâmicos, como Websphere[31], Tomcat[32] ou até o Apache. Por essa razão, a única concorrência entre

[30] http://httpd.apache.org/docs/2.4/mpm.html
[31] http://www-01.ibm.com/software/br/websphere/
[32] http://tomcat.apache.org/

o Apache e o Nginx é sobre quem vem primeiro na pilha de execução do servidor. Se é o Apache, ele sozinho consegue fazer o servidor funcionar. Se é o Nginx, ele sozinho consegue servir arquivos estáticos de forma mais rápida que o Apache (BONVIN, 2011), mas precisa de um servidor por trás para servir conteúdo dinâmico.

Uma das maiores razões de migrar do Apache para o Nginx é por causa do desempenho. O Nginx foi lançado quase uma década depois do Apache, já consolidado no mercado, e por isso teve muito tempo para estudar e entender todos os problemas e gargalos que diminuíam a eficiência do Apache. Uma das maiores qualidades do Nginx é lidar com milhares de conexões concorrentes com um único processo — como veremos mais à frente, essa é uma das vantagens de uma arquitetura assíncrona e orientada a eventos. O Apache delega cada conexão para uma *thread* ou processo diferente, sendo cada execução bloqueante para I/O, o que resulta em um *overhead* de recursos para cada conexão e limita o número de requisições que o Apache aguenta.

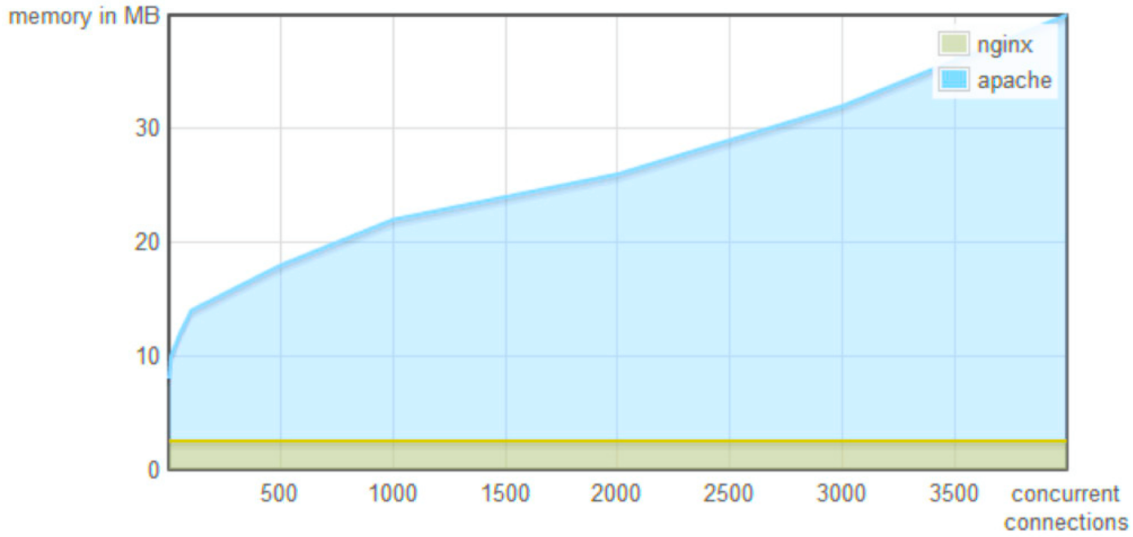
Gráfico 1: Requisições/segundo em relação ao número de conexões concorrentes.



Fonte: (D., 2008)

O Gráfico 1 mostra quantas requisições por segundo o Nginx e o Apache conseguem lidar, com relação ao número de conexões concorrentes com o servidor. Percebemos que, ao passo que o número de conexões concorrentes aumenta, o Apache, por gastar muito recurso de CPU e memória abrindo uma nova *thread* para cada conexão, passa a limitar o número de requisições que ele consegue responder por segundo, enquanto que o nginx se estabiliza.

Gráfico 2: Recursos gastos em memória em relação ao número de conexões concorrentes.



Fonte: (D., 2008)

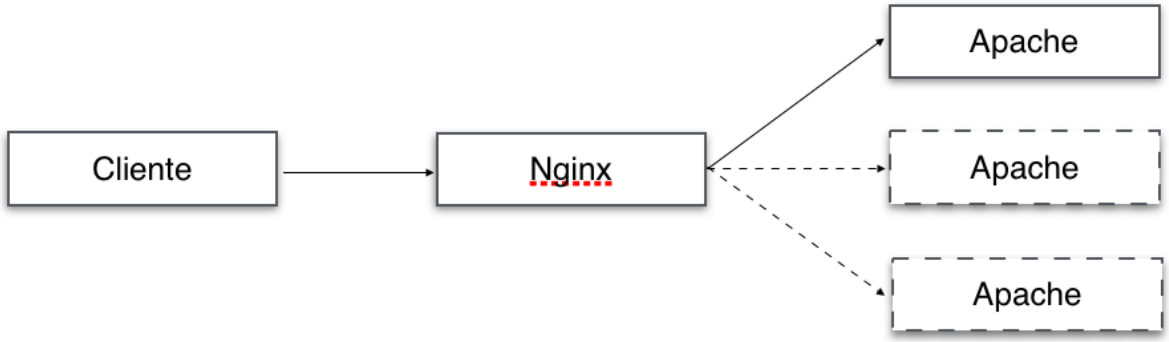
O Gráfico 2 comprova o que falamos sobre os recursos utilizados pelo Apache. Enquanto que o Nginx não aumenta o uso de memória conforme o número de conexões concorrentes cresce, — pois todas as requisições passam pelo mesmo processo — o Apache aumenta o uso dos recursos quase linearmente, o que se torna o principal motivo da diminuição de requisições por segundo.

É importante notar que a maior parte das vantagens do Nginx circula em torno de seu desempenho. O servidor Apache, devido a sua maturidade como plataforma, possui uma série de bibliotecas que podem ser carregadas dinamicamente durante sua execução e que permitem estender suas funcionalidades. O Nginx também possui um sistema de módulos, mas eles precisam ser selecionados à priori, pois são compilados no núcleo do programa. Novamente, visando o desempenho do servidor.

Ambos os servidores possuem suas vantagens e desvantagens. Como o Nginx não foi arquitetado para ser apenas um servidor web, mas também um servidor de *reverse proxy*[33], é possível unir o melhor dos dois mundos utilizando o Nginx em frente ao Apache, como ilustra a Figura 4. Essa configuração aproveita a alta velocidade de processamento do Nginx e a habilidade de lidar com inúmeras conexões paralelas. O Nginx se encarrega de servir conteúdo estático, e delega requisições de conteúdo dinâmico para o Apache, que devolve o controle ao primeiro com o recurso, e este

[33] Servidor que delega a requisição de um recurso para um ou mais servidores, com total abstração para quem requisitou.

Figura 4: Fluxograma de requisição do cliente ao servidor Web. O Nginx serve as requisições de conteúdo estático e delega as de conteúdo dinâmico para o Apache.



Fonte: O Autor

retorna a resposta ao cliente. Neste caso, o Nginx também pode acionar sua função de *reverse proxy* e realizar um balanceamento de carga para diversos servidores Apache. Isto aumenta a tolerância a falhas (pois se um servidor Apache cair, outros poderão estar disponíveis) e o desempenho da aplicação. Esta é, sem dúvida, uma das melhores configurações possíveis para um servidor Web.

Os dois modelos mencionados, LAMP e LEMP, são apenas algumas das inúmeras formas de se construir um servidor Web. Elaboramos uma seção só para elas porque, além de serem duas das mais importantes e mais utilizadas no mundo, elas descrevem genericamente o fluxo de praticamente qualquer aplicação Web. A maioria dos servidores Web segue uma arquitetura parecida, inclusive Node.js. A diferença crucial do Node.js é que ele por si só já é um servidor Web, que analisa e executa os códigos do servidor sem precisar delegar para um outro módulo, como o Apache faz. Como nos preocupamos com eficiência e escalabilidade, a plataforma implementada será intermediada pelo Nginx. Dessa forma, teremos dois sistemas assíncronos e orientados a eventos, um intermediando o outro, e aproveitaremos todas as qualidades do Nginx discutidas nesta seção. Veremos mais sobre sistemas assíncronos e orientados a eventos no Capítulo 3.

2 V8: JAVASCRIPT EFICIENTE

Antes de estudarmos a plataforma Node.js, é importante entender o software sobre o qual ela é construída: V8, o mecanismo *open source* de JavaScript do Google[34]. Esse capítulo começa explorando o que motivou o desenvolvimento do V8. Depois, explicamos as mais importantes técnicas de otimização empregadas pelo V8 para tornar o JavaScript eficiente. Finalizamos este capítulo com algumas avaliações de desempenho entre JavaScript V8 e outras linguagens de programação, além de também comparar o mecanismo V8 e outros mecanismos de JavaScript existentes.

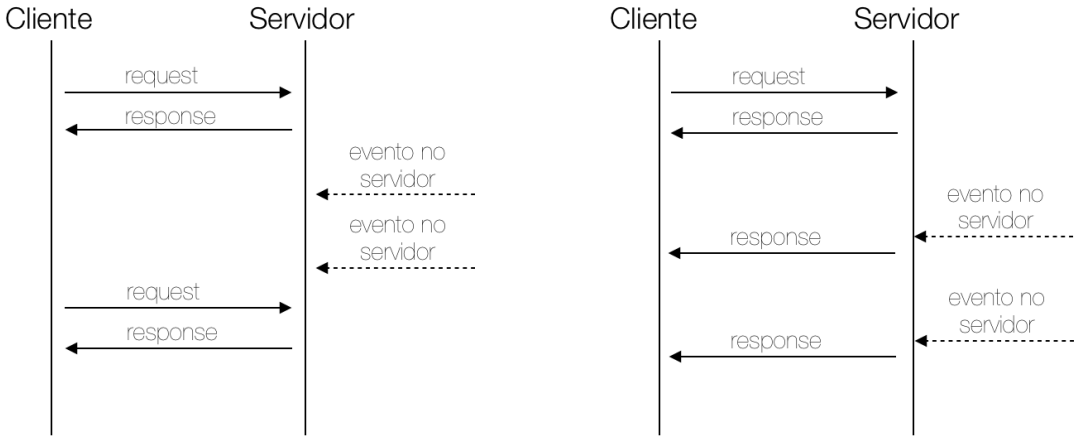
2.1 MOTIVAÇÃO

Com o surgimento do conceito de *Asynchronous JavaScript and XML* (AJAX) por Garret (2005), o horizonte de possibilidades no lado do cliente se expandiu rapidamente. Códigos JavaScript que antes ocupavam poucas linhas, agora ocupavam vários *kilobytes*. Com o AJAX, se tornou possível fazer requisições assíncronas ao servidor baseado em eventos gerados pelo usuário na própria página. Tudo isso acontecia sem necessidade de recarregar a URL. Eventualmente, surgiram ferramentas para que o próprio servidor pudesse alcançar o cliente sem o cliente precisar fazer novas requisições, tudo isso por eventos gerados no próprio servidor, como mostra a Figura 5. Esse foi o início da Web assíncrona.

Com isso, o escopo de utilidade de JavaScript cresceu de uma simples linguagem para manipular elementos da DOM[35] a uma poderosa linguagem que era capaz de mudar a forma como interagíamos com a Web. Logo tornou-se essencial para criar grandes aplicações da Web, como o Gmail[36]. Ela era eficiente no que foi projetada a fazer, mas com o grande número de ferramentas que surgiram e as novas aplicações da linguagem, ficou óbvio o gargalo no desempenho dos mecanismos de

[34] https://code.google.com/p/v8/
[35] Document Object Model é uma convenção independente de plataforma e de linguagem para representação de objetos em HTML, XHTML e XML.
[36] https://www.gmail.com

Figura 5: Diagrama de sequência de Web síncrona (esq.) e Web assíncrona (dir.)



Fonte: O Autor

execução do JavaScript, que se tornou o fator limitante no projeto de novas aplicações da Web.

Os interpretadores de JavaScript embutidos nos navegadores eram programas simples que liam e executavam o código fonte de JavaScript. Em 2008, durante a segunda guerra dos navegadores[37], e com a crescente necessidade do mercado em melhorar o desempenho do JavaScript, a corrida se tornou acirrada e otimizar a execução desses scripts se tornou um dos principais objetivos das empresas concorrentes, como Microsoft [38], Google [39] e Apple [40], com seus respectivos navegadores.

O V8 surgiu com a primeira versão pública do Google Chrome, em Setembro de 2008, como uma contra-resposta aos novos navegadores que estavam surgindo. Não demorou muito para o Google seguir a filosofia de Linus Torvalds — "dados mil olhos, todos os erros são triviais". O V8 se tornou *open source*, e em pouco tempo já havia contribuidores de todo o mundo utilizando o mecanismo e relatando erros. Além disso, tornar o V8 *open source* foi essencial para a eventual criação de plataformas baseadas nesse mecanismo, como o Node.js, que será estudado no próximo capítulo.

[37] A guerra dos navegadores é o termo utilizado para a competição entre as empresas de tecnologia pelo domínio do uso de seus navegadores Web. A primeira aconteceu na década de 90, entre o Internet Explorer da Microsoft e o navegador da Netscape, e a segunda começou há cerca de uma década entre o Mozilla Firefox, Google Chrome, Safari, Opera e Internet Explorer.
[38] http://www.microsoft.com/
[39] https://www.google.com/
[40] https://www.apple.com/

V8 foi desenvolvido no escritório do Google na Alemanha e é escrito em C++[41]. Implementa o ECMAScript e é atualmente usado para aplicações JavaScript tanto no lado do cliente, quanto no lado do servidor, embora tenha sido criado para aumentar o desempenho do JavaScript no navegador. Para ganhar eficiência, ele usa uma série de técnicas, como transformar o script em código de máquina durante a execução do programa — técnica conhecida como compilação Just-In-Time (JIT).

2.2 OTIMIZAÇÕES DO MECANISMO

O aumento na eficiência causado pelo V8 depende da natureza do seu código e de como ele é executado. Estudaremos nessa seção algumas das técnicas mais importantes empregadas pelo V8 para otimizar a execução do JavaScript. Entendê-las é importante para que o projeto seja bem implementado e aproveite o máximo da eficiência desse mecanismo. Como o Node.js é uma plataforma que funciona sobre o V8, é através desse mecanismo que ganharemos no desempenho em relação a outras tecnologias.

2.2.1 Acesso Eficiente

Em Java, C++ ou qualquer outra linguagem em que a vinculação de atributos a classes não pode ser feita dinamicamente, a localização na memória das variáveis pode ser feita em tempo constante, pois como os atributos nas classes são fixos, eles podem ser armazenados em ordem na memória. Em JavaScript, no entanto, é possível dinamicamente vincular uma nova propriedade a um objeto. Com isso, o acesso à localização das propriedades na memória depende de uma resolução dinâmica[42], o que consome ciclos na execução do código.

---

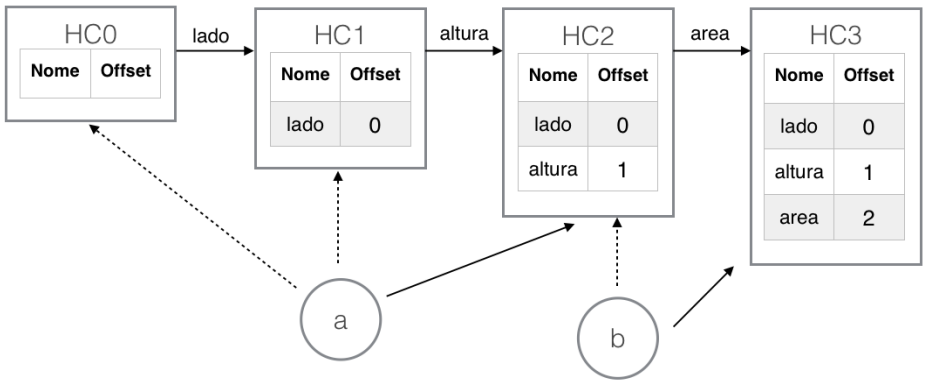
[41] http://www.cplusplus.com/

[42] Resolução dinâmica (*dynamic lookup*) consiste em resolver dinamicamente o endereço na memória das propriedades.

Para resolver esse problema, o V8 introduziu *hidden classes* (classes escondidas), conceito este estabelecido por Chambers et al. (1989). *Hidden classes* nada mais são do que uma representação alternativa dos objetos na memória. Essa nova representação faz com que o acesso às propriedades de cada objeto seja feita de forma direta, dependendo apenas de um deslocamento. Contudo, como veremos a seguir, o código deve aproveitar-se desse fato para estruturar o programa corretamente e não causar um consumo enorme de memória por parte do V8.

Figura 6: Exemplo de código JavaScript em cima e representação interna das *hidden classes* da classe Retangulo embaixo.

```
1  function Retangulo(lado, altura) {
2      this.lado = lado || 0;
3      this.altura = altura || 0;
4  }
5  var a = new Retangulo(5, 10);
6  var b = new Retangulo(1);
7  b.area = b.lado * b.altura;
```



Fonte: O Autor

Na Figura 6, vemos o que o V8 faz internamente durante a execução do código. Quando a linha 5 é executada, a declaração new ordena o V8 a criar uma nova instância do objeto Retangulo. Porém, nesse momento da execução, não há nenhuma representação dele na memória. Cria-se então a HC0 (acrônimo para *Hidden Class 0*), que não possui nenhuma entrada na sua tabela de resoluções das propriedades, e a instância a passa a apontar para ela. No entanto, dentro do construtor de Retangulo, executa-se a linha 2: this.lado = lado || 0, que instrui o programa a atribuir o valor de lado à propriedade lado ou, caso esse valor não seja definido, atribuir zero. Contudo, essa propriedade não existe em HC0. Cria-se então uma nova hidden class, HC1, que é uma extensão de HC0 com uma entrada para a nova propriedade lado, com offset 0. Esse offset é utilizado para acessar diretamente a propriedade na

memória. HC0 adiciona um ponteiro rotulado lado para HC1, significando que, da próxima vez que alguém que aponta para HC0 quiser adicionar uma propriedade nomeada lado, já existe uma: HC1. Eventualmente, a linha 3 é executada e acontece o mesmo procedimento: a passa apontar para HC2, e um ponteiro de HC1 para HC2 rotulado altura é criado. Quando b for instanciado, ele passará pelo mesmo processo de a, mas nenhuma nova hidden class será criada. Antes da linha 7 ser executada, b apontava para HC2. Contudo, a linha 7 adiciona uma nova propriedade ao objeto, que é a área do retângulo. Como essa propriedade não existe em HC2, esta adiciona um ponteiro para HC3 com o rótulo area, e b agora aponta para HC3.

Com isso, notamos que é essencial que todas as propriedades dos objetos sejam declaradas na definição inicial do objeto, e que as instâncias sempre definam estas propriedades na mesma ordem — uma hidden class com as variáveis x e y, em que xxtem offset 0 e y tem offset 1, é diferente de uma hidden class com as mesmas variáveis, mas os offsets diferentes. Os rótulos dos ponteiros de uma *hidden class* para outra também guardam os tipos das variáveis, seguindo a hierarquia de tipos do V8, que será explicada logo em breve. Portanto, devemos procurar sempre instanciar um mesmo objeto com os mesmos tipos. Por fim, além da eficiência no acesso às propriedades, o uso de *hidden classes* permite a utilização de outra técnica chamada *inline caching,* introduzida pela primeira vez em uma implementação da linguagem de programação Smalltalk-80 (DEUTSCH; SCHIFFMAN, 1984).

2.2.2 Valores marcados e arrays

JavaScript tem um único tipo de número, que é representado como um double 64 bits. Não há um tipo diferente para inteiros, então 1 e 1.0 são o mesmo valor e possuem o mesmo tipo. Internamente, o que gasta mais memória e consome mais ciclos de processamento para realizar operações aritméticas, pois essas operações com números reais de 64 bits são mais caras do que operações com números inteiros de 32 bits — e na maioria das aplicações, só estamos interessados no segundo caso. V8 “conserta" isso criando uma representação interna para inteiros. Por padrão, V8 representará qualquer tipo com 32 bits. Porém, ele utiliza um bit para marcar se é um

ponteiro para objeto genérico ou um SMI (*SMall Integer*, por ter apenas 31 bits). Se o número tem mais de 31 bits, o V8 transforma-o em um double e o encapsula em um objeto genérico. Portanto, é recomendado, sempre que puder, tentar utilizar inteiros de 31 bits com sinal.

O V8 representa arrays de duas formas diferentes: *fast elements* (elementos rápidos) e *dictionary elements* (elementos de dicionário). Se alocarmos até 100.000 elementos na memória[43], ou se instanciarmos um array vazio e inserirmos elementos densamente[44], V8 poderá representar o array com endereçamento contínuo e o acesso aos elementos será direto — e como o acesso é pelo índice, que é um inteiro, isso ainda aproveita a representação de SMI no V8 mencionada anteriormente. Se a tabela for esparsa, a representação será por meio de um dicionário, ou *map*, o que deixa o acesso um pouco mais caro. A recomendação aqui é não pré-alocar arrays grandes, e sim deixá-lo crescer à medida que precisamos de mais espaço — o V8 faz esse crescimento de forma rápida, e o acesso continua direto, desde que não ultrapassemos o tamanho mencionado. Outra recomendação é não deletarmos elementos de arrays, pois isso pode tornar o conjunto de chaves esparso.

2.2.3 Compilação de código JavaScript

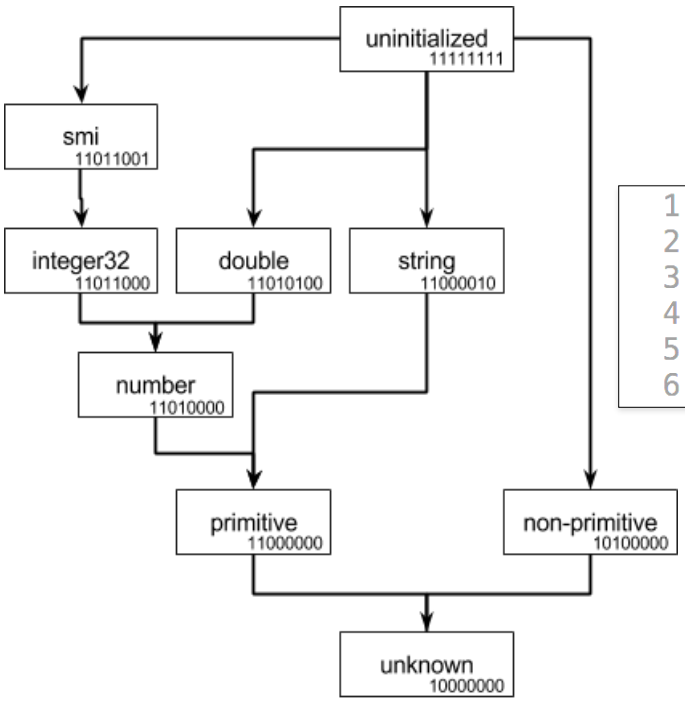
V8 compila todo o JavaScript para código de máquina assim que é executado, sem interpretadores ou geração de byte codes, como Java. No entanto, V8 tem dois compiladores: um que executa rápido e produz código genérico, e um que executa não tão rápido e produz código otimizado. O simples e rápido compilador é conhecido como *Full-Codegen Compiler* (FCC), enquanto o segundo é conhecido como *Crankshaft*. O objetivo do FCC é simplesmente receber uma árvore sintática abstrata (AST, do inglês *abstract syntax tree*)[45] como entrada e chamar o *assembler*[46] para

[43] Esse dado foi extraído do próprio código fonte do V8 disponível em https://github.com/v8/v8-git-mirror/, no arquivo /src/objects.h, linha 2260, versão 4.5.100, e é suscetível a mudanças no futuro.
[44] Chamamos uma inserção de densa quando ela preenche de forma contínua ou quase contínua, sem deixar pedaços grandes indefinidos entre valores definidos.
[45] Árvore sintática abstrata é uma representação abstrata em árvore da estrutura sintática do código fonte escrito em uma linguagem de programação.
[46] Programa que recebe instruções básicas de programação e converte em código de máquina.

gerar o código de máquina. No entanto, o FCC também otimiza o código utilizando *inline caching*, que mencionamos anteriormente.

Quando V8 percebe a existência de uma função pela primeira vez, ele analisa e a insere na AST, mas nada faz com ela. A função só é compilada quando é chamada pela primeira vez. Depois que V8 já está sendo executado, ele liga uma *thread* de *profiling* (analisador do programa) para ver quais são as *hot functions*[47]. Depois que ele teve tempo para analisar as funções, ele pode dar um *feedback* ao compilador com os tipos de cada variável utilizada na função. Esse sistema de *feedback* dos tipos durante a execução de programa é observado e armazenado pela técnica de IC. Esses tipos são representados por um byte, em que cada byte possui um prefixo que identifica o seu sucessor, e um sufixo que identifica o próprio tipo. Dessa forma, a inferência do tipo de uma variável que assume tipos distintos em diferentes execuções de uma determinada função possa ser feita em tempo constante, com uma operação *bitwise-AND*[48].

Figura 7: Representação da hierarquia de tipos no V8 na esquerda e código JavaScript na direita, em que o IC fará inferência de tipos durante a aplicação da função nas linhas 4, 5 e 6.




```
1  function sum(op1, op2) {
2      return op1 + op2;
3  }
4  var a = sum(1, 5);
5  var b = sum(3.141592, 3);
6  var c = sum("foo", 5.5);
```

Fonte: O Autor

[47] Termo utilizado para funções que são chamadas frequentemente durante a execução de um programa.
[48] Compara cada bit do primeiro operando com o bit correspondente do segundo, e se ambos os bits forem 1, o bit correspondente resultante da operação também é 1, caso contrário é 0.

Para entender melhor o funcionamento do sistema de tipos do V8, ilustramos na Figura 7 um grafo direcionado de hierarquia, no qual uma seta direcionada de um tipo A para um tipo B significa que A é um subtipo de B. A exceção a essa regra é o tipo não inicializado (*uninitialized*), que pode ser SMI, double, string ou non-primitive. Vemos também nesse grafo uma possível distribuição de byte para cada tipo. Junto a esse diagrama, disponibilizamos um código JavaScript que define uma função de soma e executa várias chamadas a essa função. Vamos analisar o que acontece quando esse código é executado:

1. Na linha 1, quando a função sum é definida, o *inline caching* do V8 grava o tipo cada um dos parâmetros e variáveis da função como uninitialized, cujo byte é 11111111.

2. Quando a função é chamada pela primeira vez, na linha 4, com sum(1, 5), o V8 sabe que ambos os valores são SMI, cujo *byte* é 11011001. Para saber o tipo de op1, faz-se o *bitwise-AND* entre 11111111 e 11011001, que é 11011001 (SMI). O mesmo ocorre para op2, que também resulta em um SMI.

3. Na segunda chamada da função, na linha 5, passa-se um *double* para op1, e op2 continua sendo SMI. O *bitwise-AND* entre o tipo de op1 (11011001) e *double* (11010100) resulta em um *number* (11010000).

4. Na última chamada da função, linha 6, uma *string* é passada para op1, e um *double* é passado para op2. O IC vai agora fazer o *bitwise-AND* entre o tipo de op1 (11010000) e *string* (11000010), que resulta em um *primitive* (11000000). O tipo de op2 passa pelo mesmo processo que op1 passou no passo 3, e gera um *number*.

No final da execução, o IC sabe que op1 é um *primitive*, enquanto op2 é um *number*. Todas essas informações são gravadas nos nós da AST associadas à função. Durante a execução do V8 e do *profiler*, quando ele já identificou uma *hot function* e o IC coletou feedback de tipos, ele tenta rodar a AST da função por meio do *Crankshaft*. Este compilador, por fim, compila a AST utilizando uma série de técnicas de otimização, como *inlining*, valores marcados (que foi descrito anteriormente), inferência de tipos (por meio de IC), além de várias outras técnicas que são descritas por Conrod (2013).

2.2.4 Garbage Collection Eficiente

O ECMAScript, no qual o V8 é baseado, não especifica nenhuma interface para Garbage Collection (GC) — o processo de recuperar a memória ocupada por objetos que não são mais utilizados. Sistemas de GC negligenciados tendem a levar o programa a grandes interrupções, comprometendo fortemente o desempenho do mesmo.

Afirmamos que um objeto está vivo na memória quando é apontado por um objeto raiz ou algum outro objeto vivo. Quando eles estão mortos, eles ocupam espaço desnecessário na memória, que poderia ser utilizado por objetos vivos. Os objetos raiz são apontados diretamente pelo V8, e eles são vivos por definição. O objetivo do GC é, então, identificar os objetos mortos e liberar a memória utilizadas por ele.

O GC do V8 não possui efeitos colaterais, é preciso e geracional. Quando um ciclo de GC começa, o V8 interrompe a execução do programa; mas, para minimizar o impacto da interrupção, ele processa apenas parte do *heap* de objetos na maior parte dos ciclos. Na maioria dos programas, objetos tendem a morrer cedo. O V8 tira vantagem desse fato dividindo o *heap* de objetos em duas partes: um espaço novo no qual objetos são criados, e um espaço antigo para onde objetos que sobreviveram a ciclos de coleta anteriores são promovidos. O espaço novo geralmente é pequeno, possuindo de 1MB a 8MB de memória[49].

Durante a execução do programa, eventualmente a memória disponível para o espaço novo pode acabar; nesse caso, um ciclo menor de GC é acionado para remover os objetos mortos do espaço novo. Objetos que sobreviveram aos dois últimos ciclos menores de GC são promovidos para o espaço antigo. Quando uma determinada quantidade de memória é promovida do espaço novo para o espaço antigo, um ciclo maior de GC é acionado para reciclar este espaço antigo. Esses ciclos maiores são mais raros, pois como foi dito, a maioria dos objetos morrem cedo, e o ciclo menor de GC já é suficiente para manter memória disponível no espaço novo e não compromete drasticamente na eficiência do programa.

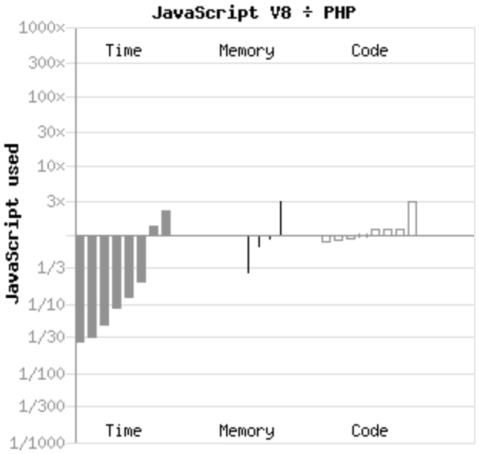
[49] A quantidade de bytes alocado para o espaço novo depende das heurísticas utilizadas no V8 para o programa executado.

2.3 AVALIAÇÕES DE DESEMPENHO

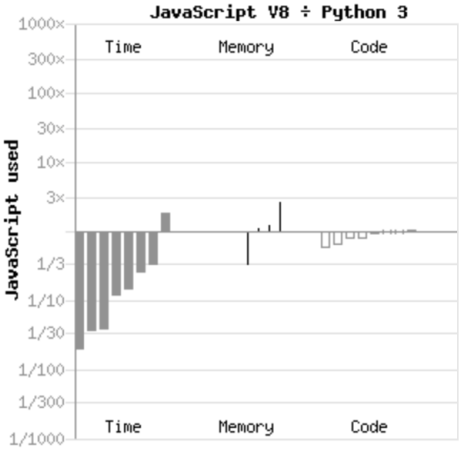
Realizamos uma série de comparações de desempenho entre JavaScript V8 e PHP, Python 3 e Ruby, linguagens frequentemente usadas no desenvolvimento Web. Também comparamos o desempenho do V8 com C++. É bom lembrar que as linguagens de programação foram comparadas como se fossem projetadas para o mesmo propósito — o que comumente não são. JavaScript e C++ possuem espectros de aplicações bastante distintos, mas a comparação entre os dois nos fornece uma introspecção de onde estamos e o que ainda podemos alcançar em questões de eficiência.

Gráfico 3: Comparações entre JavaScript V8 e outras linguagens de programação. Retângulos e linhas abaixo do eixo principal representam um melhor desempenho do JavaScript em relação à tecnologia comparada, e vice-versa.

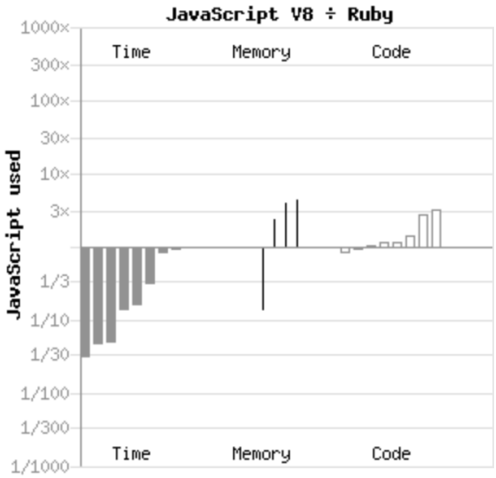
a) JavaScript V8 vs PHP.



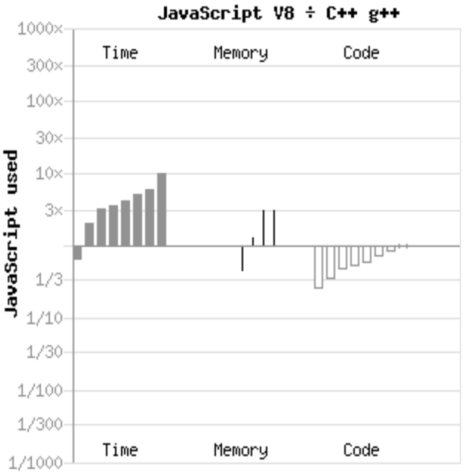
b) JavaScript V8 vs Python 3.



c) JavaScript V8 vs Ruby.



d) JavaScript V8 vs C++.



Fonte: Adaptado de (THE COMPUTER LANGUAGE BENCHMARKS GAME)

As métricas de desempenho avaliadas foram tempo de execução, memória utilizada e quantidade de linhas de código. Todos os testes foram feitos em um sistema Ubuntu[50] x64, processador Intel Q6600 de um núcleo. Os *benchmarks* e resultados gerados nessa arquitetura foram extraídos do The Computer Language Benchmarks Game[51]. Todos os resultados comparam a execução mais rápida de um *benchmark* em uma linguagem vs. a execução mais rápida do mesmo *benchmark* em outra linguagem.

Cada retângulo ou linha dos Gráficos 3a, 3b, 3c e 3d representa um benchmark diferente para uma métrica diferente, de forma que o mesmo algoritmo foi implementado nas linguagens comparadas. Nos testes realizados, para as mesmas tarefas, JavaScript V8 rodou em média 13.9 vezes mais rápido que PHP, onde em 6 dos 8 testes, JavaScript foi mais rápido, e no pior caso ele ainda assim não chegou a ser duas vezes pior do que PHP; contra Python, ele foi em média 15.8 mais rápido, onde em 7 dos 8 testes ele teve uma melhor performance, e no caso ruim ele também não chegou a ser muito pior; ele não teve nenhum desempenho pior do que Ruby, mas pelo contrário, ganhou no tempo de execução em todas, executando em média 11 vezes mais rápido. Por fim, C++ rodou em média 4.3 vezes mais rápido que o JavaScript V8, o que não é ruim, tendo em vista a diferença de paradigma das duas linguagens. Por C++ ser estaticamente tipada e suas classes não serem dinamicamente extensíveis, a linguagem não precisa de um compilador otimizador durante o tempo de execução, pois a maioria das otimizações podem ser feitas diretamente na primeira compilação; já V8, como discutimos, tem todo o *overhead* de troca de compiladores e técnicas de otimização de baixo-nível para superar as desvantagens de ser uma linguagem dinamicamente tipada e interpretada.

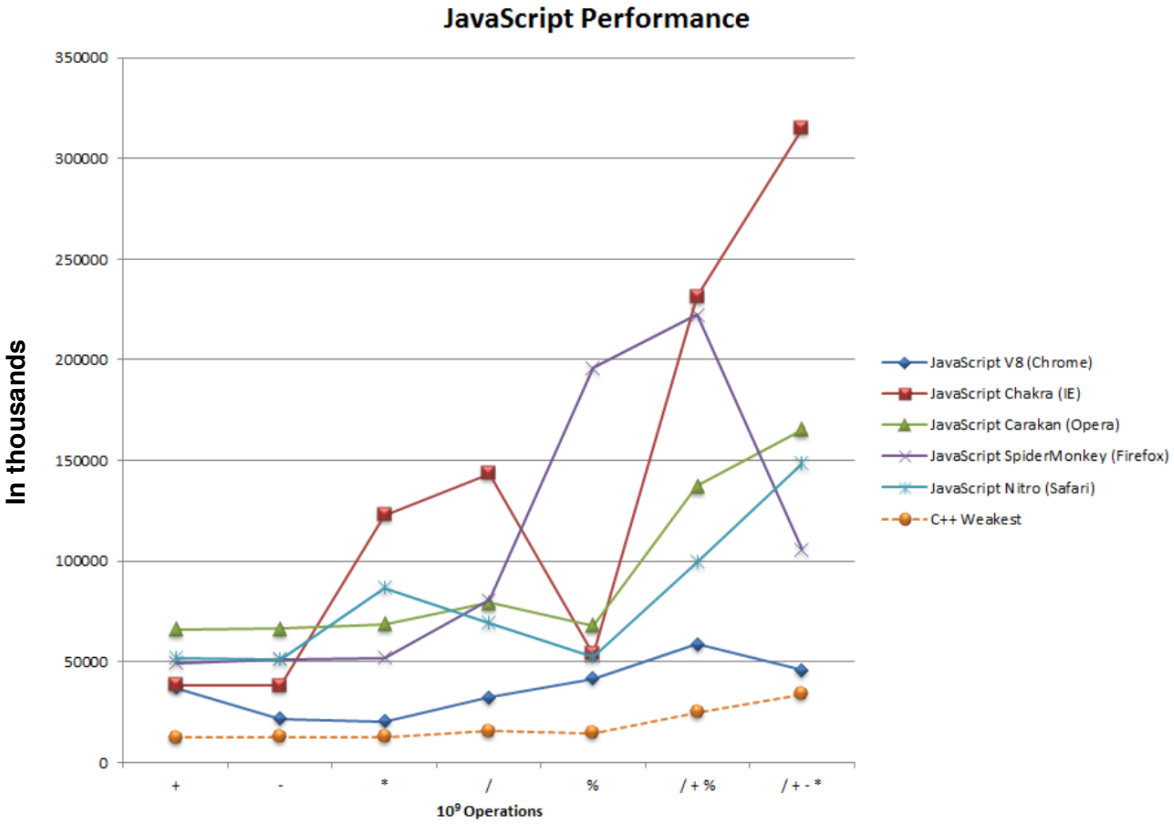
Em relação à memória utilizada, JavaScript aloca em média a mesma quantidade que PHP e Python, e um pouco mais que Ruby e C++, embora a diferença não seja significativa. A diferença também não é grande para a relação entre as linhas de códigos utilizadas para escrever o mesmo programa em JavaScript, PHP e Python. Ruby consegue expressar a mesma tarefa com menos linhas de código, embora a diferença não seja significativa; no entanto, JavaScript V8 compensa isso tendo um melhor desempenho na execução em todos os *benchmarks.* Por fim, nos testes

[50] http://www.ubuntu.com/
[51] http://benchmarksgame.alioth.debian.org/

realizados, C++ sempre exigiu mais linhas de código do que JavaScript para realizar a mesma tarefa. C++ é focado em tornar a vida mais fácil para o computador, de forma a aumentar o desempenho, enquanto JavaScript está disposto a sacrificar um pouco do desempenho para facilitar a vida do programador. Isso torna escrever um programa rápido em C++ altamente dependente do esforço e conhecimento do programador, o que pode às vezes refletir em mais linhas de código para executar uma simples ação. Enquanto isso, as técnicas de otimização empregadas pelo V8 justificam por si só a eficiência do JavaScript V8, o que tira o peso do programador de se preocupar com detalhes mínimos de otimizações de alto nível e o permite atribuir mais esforço na lógica e em testes da aplicação.

Por fim, no Gráfico 4 é mostrado uma importante comparação entre a eficiência do mecanismo V8 com os mecanismos de JavaScript de outros navegadores (RAPPL, 2012). Além disso, o desempenho do C++ sem as otimizações O2 e O3 é utilizada como base de comparação. Um bilhão de operações aritméticas foram realizadas em cada uma das linguagens.

Gráfico 4: Comparação entre C++ e mecanismos de JavaScript para operações aritméticas.



Fonte: (RAPPL, 2012)

Percebemos que o V8 obteve, nos testes realizados, um desempenho melhor do que os outros mecanismos de JavaScript. A diferença entre JavaScript V8 e C++ já era esperada, como discutimos anteriormente. Esses resultados fortalecem o que discutimos extensivamente nesse capítulo: V8 é um dos mecanismos mais eficiente de JavaScript existente atualmente, e plataformas criadas sobre ele, como o Node.js, aproveitam do seu desempenho.

2.4 CONSIDERAÇÕES FINAIS

Apesar do papel do V8 para o programador ser obstruído pelos componentes de mais alto-nível do Node.js, que conheceremos no próximo capítulo, é a partir desse mecanismo que toda a plataforma opera. Com isso em mente, entender as técnicas de otimização empregadas pelo V8 é fundamental para a construção de um projeto que aproveite o máximo possível os benefícios de desempenho providos por ele.

O mecanismo V8 dá ao JavaScript, linguagem dinamicamente tipada e interpretada, um poder de desempenho que não é comum em linguagens que seguem os mesmos paradigmas. Entendemos nesse capítulo as principais técnicas de otimização empregadas pelo mecanismo, cuja eficácia se torna evidente nos testes de avaliação de desempenho efetuados.

Para mais informações, um estudo mais aprofundado dos *benchmarks* de JavaScript V8, no nível de arquitetura de hardware, é feito por Tiwari e Solihin (2012) em *Architectural Characterization and Similarity Analysis of Sunspider and Google's V8 Javascript Benchmarks*.

3 NODE.JS

Estudamos até aqui os acontecimentos e mecanismos que fizeram a existência do Node.js possível. Com essa base, este capítulo explica sobre como o Node.js surgiu, quais suas características gerais que a destaca de outras tecnologias e como a plataforma é arquitetada; terminamos com os resultados de *benchmarks* que preparamos para comparar a eficiência do Nginx e Node.js, conjunto que utilizaremos na implementação do projeto, com o Apache e PHP, que é um dos arquétipos mais utilizados no desenvolvimento Web.

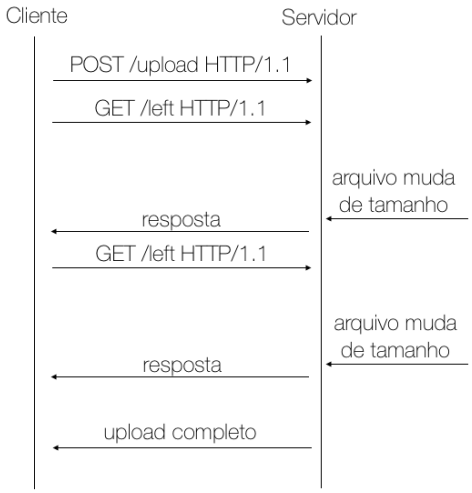
3.1 CONTEXTO HISTÓRICO

Em 2009, no ano seguinte ao lançamento do Google Chrome e a disponibilização pública do V8, Ryan Dahl estava tentando resolver um problema até então difícil: fazer o navegador saber quanto tempo restava em um processo de upload. Uma técnica bastante utilizada era enviar uma requisição AJAX a cada determinado intervalo de tempo para perguntar ao servidor quanto do arquivo já tinha sido enviado. Mas isso consumia recursos do servidor, porque uma nova requisição era estabelecida constantemente para fazer uma pergunta cuja resposta poderia às vezes ser a mesma, repetidamente. Uma alternativa a esse método era *long polling*, que consistia em fazer uma pergunta ao servidor, mas este segurava a resposta até que algum evento acontecesse — nesse caso, quando o arquivo mudasse de tamanho. Esse processo é mostrado na Figura 8.

Dahl tentou implementar um servidor que englobasse esse conceito em várias linguagens, como Ruby, Lua[52] e Haskell[53]. O problema encontrado por ele é que essas linguagens tinham natureza síncrona, e a criação e gerenciamento de *threads* fugia do escopo de simplicidade que ele buscava, além de demandar um custo de memória e CPU muito alto para cada conexão nova. Dahl buscava uma linguagem de

[52] http://www.lua.org
[53] https://www.haskell.org

Figura 8: Diagrama de sequência de *long polling*, onde a API do servidor é: /upload para carregar arquivos; /left para saber quanto já foi carregado do arquivo em questão.



Fonte: O Autor

programação em que fosse simples realizar operações assíncronas, para que a CPU não ficasse ociosa em cada operação de I/O demorada, como leitura e e escrita de arquivos grandes. Por mais que a maioria das linguagens de programação testadas por ele fornecessem bibliotecas e meios de se implementar assincronismo, nenhuma fazia isso no nível de linguagem — isto é, elas eram síncronas por natureza, mas poderiam ser assíncronas por extensão.

Nessa mesma época, várias empresas que queriam dominar o mercado dos navegadores estavam investindo tempo e dinheiro em uma linguagem em particular: JavaScript. Com isso, o esforço dessas empresas em aumentar o desempenho de execução dessa linguagem já era visível em alguns projetos *open source*. JavaScript tinha todas as qualidades que Ryan Dahl estava buscando em uma linguagem de programação, além de trazer inúmeros outros benefícios quando empregada no lado do servidor. Ela era simples e leve, o que a permitia executar em sistemas com especificações mínimas de hardware; tinha suporte a operações não-bloqueantes, o que permitia ao servidor liberar o uso da CPU quando uma operação assíncrona fosse executar; executava em uma única *thread*, o que tirava o peso do gerenciamento de uma *thread pool*, diminuía o uso de memória e CPU e ainda assim simulava paralelismo; e era orientada a eventos, o que se encaixava perfeitamente nos requisitos para implementação do *long polling*. O único problema, a eficiência, era resolvida pelo mecanismo V8, que era *open source*. Dessa forma, ficou óbvio para

Dahl que ele utilizaria essa linguagem para implementar seu servidor Web. Conforme o projeto foi crescendo, e um padrão foi surgindo, a plataforma que ele tinha desenvolvido era agora um servidor Web por completo: Node.js.

3.2 CARACTERÍSTICAS GERAIS

Node.js é uma plataforma assíncrona, orientada a eventos, para desenvolvimento de aplicações de rede escaláveis. Como tal, possui uma série de características que a diferem de outras arquiteturas de desenvolvimento Web. Nesta seção, estudamos as principais características e benefícios do Node.js que tornam essa plataforma única e poderosa.

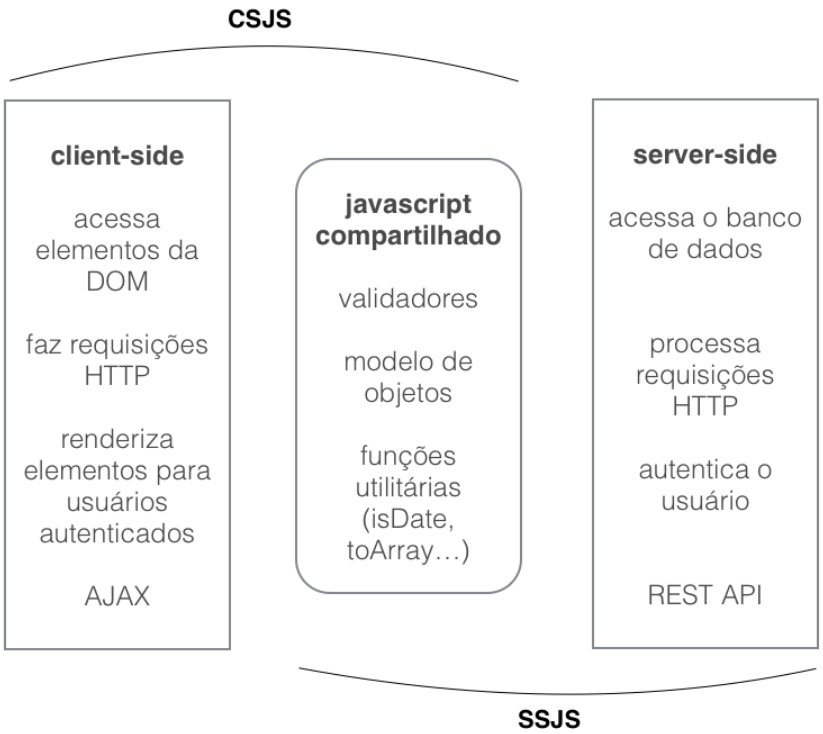
3.2.1 Diferente de JavaScript Tradicional

Apesar de ter sido modelada a partir do ECMAScript, Node.js e JavaScript de navegador diferem no quesito de variáveis globais (MARDAM, 2012, p. 14). Quando a *keyword* var é omitida, o JavaScript tradicional aloca a variável no escopo global, poluindo o mesmo. Isso é citado por Crockford (2008) como uma das partes ruins de JavaScript. Já no Node.js, qualquer variável pertence ao escopo no qual foi declarada por padrão. No JavaScript de navegadores, é muito difícil manter um gerenciamento de dependências, porque os scripts utilizados são definidos em uma linguagem de marcação (HTML) e as variáveis globais de cada script são misturadas em um único escopo, durante a execução, comprometendo variáveis globais com o mesmo nome. Em JavaScript, o gerenciamento de dependências é feito na própria linguagem, por meio das palavras reservadas module.exports e exports. Além dessas duas palavras reservadas, Node.js define dois novos objetos globais: process e global, que substituem o uso de window. global se refere às variáveis globais do código, enquanto process se refere às variáveis relativas ao processo em execução como, por exemplo, o diretório no qual o processo foi executado e as variáveis de ambiente.

3.2.2 Compartilhamento de Código

Client-Side JavaScript (CSJS) é nome que se dá ao código dessa linguagem que executa no lado do cliente para acessar elementos da DOM, fazer chamadas assíncronas ao servidor, renderizar elementos dinâmicos, calcular valores, entre outras coisas. Server-Side JavaScript (SSJS) é o código JavaScript que executa no lado do servidor. É responsável por acessar o banco de dados, atender a requisições, autenticar o usuário e fazer tudo o que um servidor Web faz. Node.js é uma plataforma SSJS.

Figura 9: Exemplo de funções exercidas pelo JavaScript do cliente e do servidor.



Fonte: O Autor

Arquitetar um projeto que utiliza a mesma linguagem no servidor e no cliente possui uma série de vantagens. Em termos de código compartilhado, vemos na Figura 9 que, por mais que o JavaScript em cada um dos lados tenha suas funções um tanto polarizadas, encontramos um meio termo em que muito código pode ser reutilizado

nos dois lados, como validadores de campos[54], modelagem dos objetos e inúmeras funções utilitárias. Outra vantagem é que quando precisamos aprender várias tecnologias para projetar todo o sistema (para o *backend* e o *frontend*), às vezes acaba comprometendo todo o projeto. Na arquitetura proposta, JavaScript é a única linguagem de programação envolvida no sistema — a menos que este cresça e passe a envolver conceitos que requeiram outras tecnologias. Por fim, JavaScript lida naturalmente com *JavaScript Object Notation* (JSON) {ref.}, que é bastante utilizada para troca de dados entre sistemas diferentes. Vários bancos de dados já retornam o resultado de pesquisas em JSON, como o MongoDB[55], que já é naturalmente utilizada para representar objetos em JavaScript. Essa interoperabilidade não é muito comum em outras linguagens. Quando temos uma mesma representação de dados nas várias camadas de desenvolvimento, removemos uma boa parte do *overhead* de conversão desses dados.

3.2.3 Assincronicidade

Para entender como operações de leitura e saída são feitas de forma não-bloqueante em Node.js, precisamos saber o que é *Event Loop*[56]. Tudo em Node.js gira em torno do *Event Loop*: é a partir disso que as ações são executadas na plataforma. Para entender esse conceito, vamos analisar a execução do código na Figura 10, que executa a leitura do arquivo foo.txt.

Figura 10: Leitura assíncrona de arquivos em JavaScript com o uso de *callbacks*.

```
1  fs = require('fs');
2  fs.readFile('./foo.txt', 'utf-8', function callback(err, data) {
3    if (err) {
4      return console.log(err);
5    }
6    console.log(data);
7  });
```

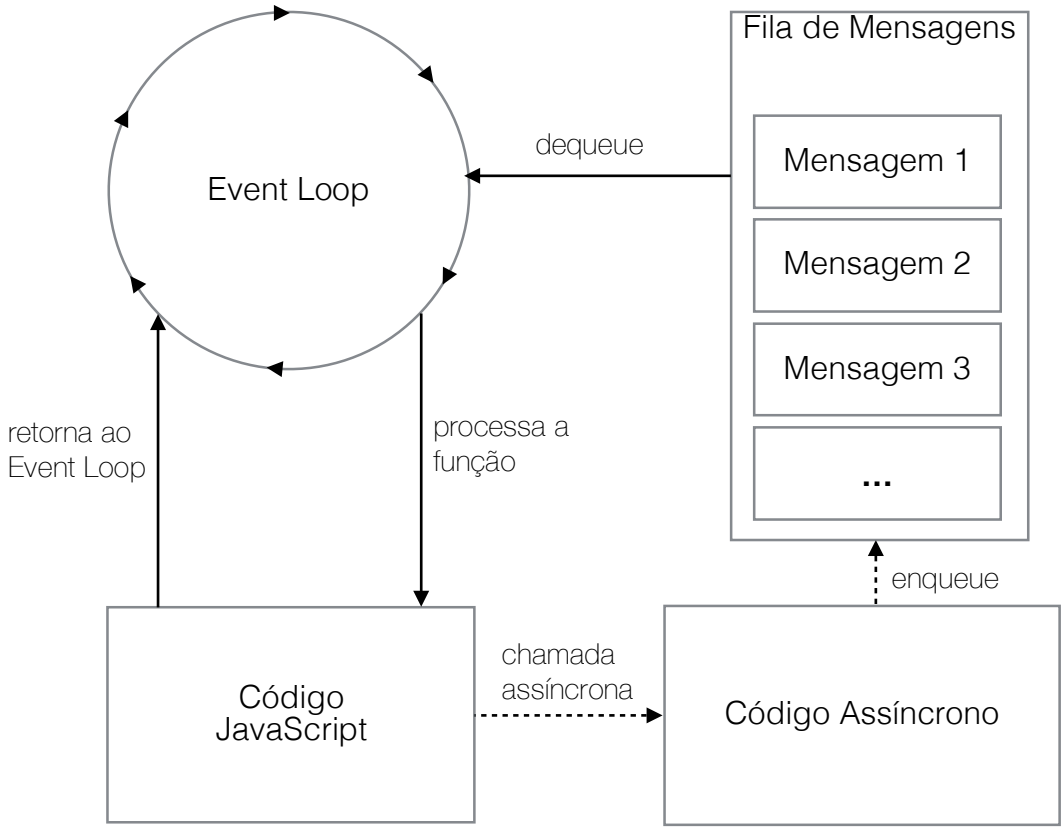
Fonte: O Autor

[54] Muitos formulários online disponibilizam um feedback dinâmico para os campos (se são válidos ou inválidos). Como um usuário malicioso pode desativar as verificações no cliente, elas devem ser feitas novamente no lado do servidor.
[55] https://www.mongodb.org/
[56] https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

A linha 1 importa o módulo fs, que lida com operações de entrada e saída. Na linha 2, chamamos a função readFile de fs, que recebe no primeiro parâmetro o caminho do arquivo, no segundo parâmetro o tipo de formatação e no terceiro parâmetro uma função de *callback* que será chamada quando a leitura do arquivo terminar. Todos os objetos em JavaScript são tratados como *first-class citizens*[57], incluindo as funções, o que possibilita passar uma função como argumento. Quando a leitura do arquivo é concluída, a função de *callback* é chamada com dois parâmetros. O primeiro parâmetro é nulo se não tiver ocorrido nenhum erro durante a leitura, caso contrário ele contém o erro. Se não houver erros, o segundo parâmetro é o *buffer* com o arquivo lido. Veremos agora como o Node.js lida com isso internamente.

Node.js possui uma fila de mensagens a serem processadas. Cada mensagem reflete o evento que a acionou e a função de *callback* que foi passada no registro desse evento. Essas mensagens são colocadas na fila de espera em decorrência de

Figura 11: Ciclo de funcionamento interno do Node.js.



Fonte: O Autor

[57] Tipos que são *first-class citizens* em linguagens de programação podem ser passados como argumento de funções, usados como valor de retorno, e atribuídos à variáveis ou armazenados em estruturas de dados.

eventos, que podem ser tanto uma nova conexão HTTP como o retorno de uma operação de leitura de um arquivo feita pelo sistema operacional. Quando essas mensagens chegam na frente da fila, elas são logo processadas. Logo, esse processo é sequencial, e nada até agora sugere a famigerada assincronicidade de que tanto falamos. O funcionamento do Node.js é ilustrado na Figura 11.

Quando o código JavaScript está sendo processado, existem chamadas que podem ser delegadas para um módulo à parte. Quando o Node.js, durante a execução da função, detecta operações assíncronas, como o fs.readFile, ele delega o trabalho de executar essa função para outro módulo, que conheceremos mais à frente quando entendermos a arquitetura da plataforma. Esse módulo, por sua vez, é responsável por criar *threads* para a execução dessas tarefas. Quando cada tarefa é completada, a função de callback é transformada em uma mensagem e enfileirada na fila de mensagens, para que o Event Loop eventualmente a execute. Enquanto isso, o código JavaScript original que estava executando continua sua execução com a linha seguinte à chamada da função assíncrona. No término da função, o controle é passado de volta ao Event Loop. No caso do código exemplificado pela Figura 10, tão logo o programa executasse a linha 2, o sistema operacional tomaria conta de ler o arquivo em uma *thread* dedicada, e o controle seria passado novamente para o *Event Loop* — pois não há nenhuma linha de código para ser executada depois. Assim que o sistema operacional termina de ler o arquivo foo.txt, a função *callback* é colocada na fila de mensagens, e logo será escalonada para execução pelo *Event Loop*. Quando isso acontecer, os dados do arquivo (ou erro, se houver) serão mostrados na tela, e nenhuma função assíncrona adicional será chamada para execução. Dizemos que esse processo simula o paralelismo para operações de I/O pois *threads* são criadas para execução dessas tarefas em paralelo, com total abstração para o programador.

3.2.4 Servidor Integrado

Diferente de PHP, Python, Ruby, Java e outras linguagens de programação, nas quais o desenvolvimento Web requer configurar um servidor externo para funcionar, Node.js por si só já traz um servidor Web embutido. E diferente do Django, *framework*

de desenvolvimento Web para Python, que traz um servidor não-escalável para produção, o Node.js consegue dar suporte a um sistema de porte razoável sozinho, lidando bem com centenas à milhares de conexões concorrentes, dependendo do sistema em que o Node.js é executado. Para grandes sistemas é facilmente escalável, dado que tem acesso a mais processamento, mais memória e servidores de *reverse proxy*, como Nginx, que é facilmente configurável e integrável com o Node.js. O servidor nativo do Node.js traz um benefício a pessoas que desejam aprender desenvolvimento Web, pois funciona com poucas linhas de código e não requer configuração de software externo, como mostra a Figura 12.

Figura 12: A criação do servidor Web é feita dentro do próprio código JavaScript. Nesse código, um servidor é criado na linha 2 para servir todas as requisições na porta 1337 do servidor local (linha 5), respondendo com sucesso qualquer requisição com “Hello World”.

```
1  var http = require('http');
2  http.createServer(function (req, res) {
3    res.writeHead(200, {'Content-Type': 'text/plain'});
4    res.end('Hello World\n');
5  }).listen(1337, '127.0.0.1');
```

Fonte: O Autor

O módulo http, que abstrai as funções da interface de rede, provê ao desenvolvedor um modo de integrar o servidor Web e a lógica da aplicação em um único modelo de sistema. Dessa forma, fica a cargo do programador as peculiaridades do servidor Web, como *cache*, análise de *cookies* e outras funções que a maioria dos softwares externos de servidor não mistura com a aplicação. Isso dá ao programador o benefício de poder configurar os detalhes funcionais do servidor Web com o mesmo conhecimento tecnológico que possui para programar a aplicação.

3.2.5 Gerenciador de Pacotes

Node Package Manager (NPM)[58], da tradução literal do inglês, é o gerenciador de pacotes do Node.js. Um conjunto de funcionalidades pode ser encapsulado em um pacote, e enviado ao registro do NPM, para que outros desenvolvedores possam

[58] https://www.npmjs.com/

reutilizar o código. Todos os projetos de Node.js possuem um arquivo package.json, que define as dependências do projeto para outros módulos. Dessa forma, há um compartilhamento enorme de código entre a comunidade de Node.js. A quantidade de projetos em JavaScript no GitHub, famoso serviço de hospedagem de repositórios Git online, é maior do que qualquer outra linguagem[59].

É muito difícil encontrar um projeto de Node.js que não use ao menos um pacote de dependência. A plataforma tenta ser o mais simples e leve possível, atribuindo ao programador a responsabilidade de encontrar módulos que são necessários para o funcionamento do projeto, ou criá-los caso eles não existam — e se possível, disponibilizar para que outros projetos possam reutilizar. O objetivo do NPM é reduzir a quantidade de código *boilerplate* do projeto, de forma que os desenvolvedores tenham inúmeras opções de bibliotecas que simplifiquem a estrutura da aplicação. Por exemplo, se criamos um servidor Web utilizando apenas as bibliotecas nativas do Node.js, utilizaremos muitas linhas de código e lógica a mais do que se utilizarmos *frameworks* Web disponíveis no NPM, como o *express.js*, que além de facilitar o gerenciamento do servidor e utilizar muito menos linhas de código, cria uma ótima estrutura lógica de desenvolvimento.

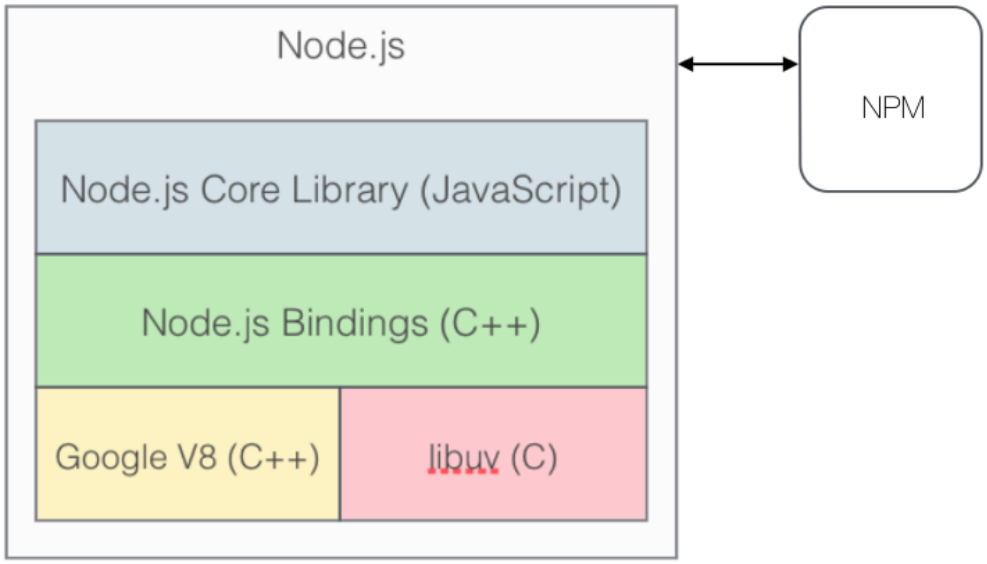
3.3 ARQUITETURA

Node.js estende as funcionalidades do JavaScript V8 para poder interagir com o sistema operacional: escrever em arquivos, ler de arquivos, fazer operações de redes, criar subprocessos, entre outros. Todas as operações de baixo nível, que lidam com o sistema operacional, são feitas de forma assíncrona, por mais que o programa execute em uma única *thread*. Para entender como isso acontece, precisamos saber como a plataforma é arquitetada (Figura 13).

No fundo da pilha estão o V8 e a biblioteca libuv. Essa biblioteca fornece a assincronicidade e o poder de operações baixo-nível da plataforma: operações de leitura e saída; interface de rede — conexão via TCP ou UDP, resolução de DNS[60],

[59] Informação extraída em maio de 2015 de http://githut.info.
[60] *Domain Name System* é um sistema de nomes distribuído para computadores, serviços ou qualquer recurso conectado à Internet ou a uma rede privada.

Figura 13: Arquitetura simplificada do Node.js com o NPM.



Fonte: O Autor

entre outros; bifurcação de processos; e o próprio *event loop*, que coordena como o Node.js funciona. Portanto, o libuv coordena toda a parte de rede, sistema de arquivos e “paralelismo" do Node.js. Toda vez que uma operação de natureza assíncrona é chamada, ela é passada para o libuv, que internamente gerencia uma *thread pool* para execução dessas tarefas, com total abstração para o programador.

Acima do V8 e libuv está a camada de “colagem” — *Node.js Bindings*, escrita em C++, que une essas duas tecnologias junto ao sistema operacional, para que funcionem harmoniosamente. Acima dessa camada, todo código é escrito em JavaScript. Os módulos principais do Node.js são incluídos na biblioteca padrão da plataforma — Node.js Core Library. São eles:

- **http** - módulo responsável pelo servidor HTTP do Node.js;
- **util** - provê funções utilitárias, como depuração, formatação, verificação de tipos, entre outras;
- **querystring** - como o próprio nome sugere, implementa funções de query string;
- **url** - responsável por prover funções utilitárias para análise e manipulação de URL;
- **fs** - lida com operações do sistema de arquivo, como leitura e escrita;
- **crypto** - funções de criptografia;

- **path** - lida com o caminho no sistema de arquivos do sistema operacional em que a plataforma está rodando.

Acima de todas essas camadas executa a aplicação, a qual utiliza os módulos de alto nível do Node.js para executar o servidor Web. É importante mencionar que o gerenciador de pacotes do Node.js, NPM, executa ao lado dessa arquitetura para prover ao desenvolvedor módulos que implementam inúmeras funcionalidades e não obrigá-lo a "reinventar a roda".

3.4 BENCHMARKS

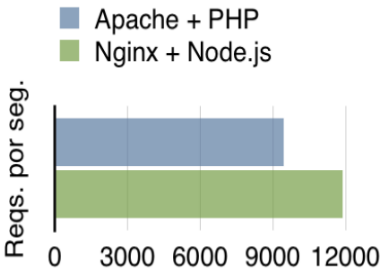
Escolhemos a configuração Web mais utilizada atualmente, Apache e PHP, para comparar com a arquitetura estudada. Em todos os cenários, utilizamos o Nginx como *reverse proxy* do servidor Node.js, e o Apache como servidor Web de PHP. Os *benchmarks* implementados foram testados com a ferramenta ApacheBench[61], no sistema operacional OS X Yosemite[62], 1 processador Intel Core i5 2.4 GHz com 2 núcleos. Para cada *benchmark*, executamos 20.000 requisições no total com 100 requisições concorrentes por vez.

3.4.1 Arquivos Estáticos

Para requisição de uma página de 50 bytes, a configuração AP (Apache + PHP) obteve uma média de requisições por segundo menor do que a configuração NN (Nginx + Node.js), como podemos perceber pelo Gráfico 5. Todos os arquivos estáticos são servidos pelo Nginx, o qual é altamente otimizado para esse tipo de requisição. Por mais que servir arquivos estáticos rapidamente seja tão importantequanto servir outros tipos de arquivos, este teste não avalia o poder de processamento do Node.js.

[61] http://httpd.apache.org/docs/2.2/programs/ab.html
[62] https://www.apple.com/osx/

Gráfico 5: Comparação entre Apache + PHP e Nginx + Node.js para servir arquivos estáticos.



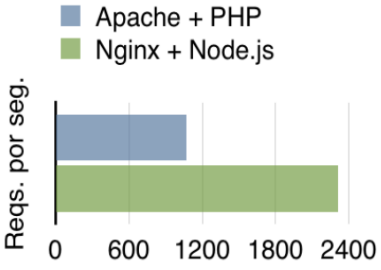
Arquivos Estáticos	Média req / s	Média tempo / req
Apache + PHP	9429.12	10 ms
Nginx + Node.js	11868.44	8 ms

Fonte: O Autor

3.4.2 I/O bound

I/O bound refere-se à condição na qual a maior parte do tempo de computação é dedicada a operações de entrada e saída. Nesse teste, simulamos uma operação de leitura que demora 10 ms nas duas configurações — não fizemos a leitura propriamente dita para não correr o risco de utilizar uma operação de leitura de arquivo que seja mais otimizada em uma linguagem do que em outra. O Nginx delegará a operação para o Node.js, por não ser uma requisição de arquivo estático. Como são operações de I/O, o Node.js chama o libuv para processá-las assincronamente, mas as requisições executam em uma única *thread*. Enquanto isso, o Apache despacha cada requisição para um recurso dedicado na CPU, o que acarreta em mais ciclos de CPU para criação de *thread* e trocas de contexto. O Gráfico 6 mostra os resultados.

Gráfico 6: Comparação entre Apache + PHP e Nginx + Node.js com foco em operações de I/O.



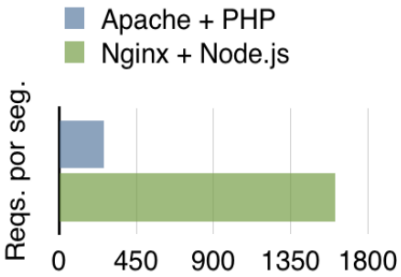
I/O	Média req / s	Média tempo / req
Apache + PHP	1068.84	95 ms
Nginx + Node.js	2312.92	43 ms

Fonte: O Autor

3.4.3 CPU bound

CPU bound refere-se à condição na qual a maior parte do tempo de computação é ocupado por processamento na CPU. Para esse teste, implementamos o algoritmo trivial de multiplicação de matrizes $O(n^3)$ em PHP e Node.js, e cada requisição executava esse algoritmo com duas matrizes de tamanho 50 X 50. O Apache, utilizando MPMs, delega cada requisição para subprocessos. No entanto, o Node.js executa todas as requisições sequencialmente, o que dá indícios de que ele demoraria mais para terminar de processar todas as requisições. Contudo, há dois motivos principais para que o NN tenha sido muito mais eficiente que o AP: em primeiro lugar, como vimos na seção de avaliações de desempenho do Capítulo 3, o V8 executa operações muito mais rapidamente que o interpretador de PHP; segundo, pela mesma razão de Node.js ter sido melhor que PHP em termos de I/O bound: a abordagem de criação de *threads* para cada requisição resulta no *overhead* de tempo de criação por *thread*, troca de contexto, número de conexões nunca poder ser maior que o número de *threads*, entre outros. Tudo isso acaba tornando essa técnica muito mais pesada e demorada, como podemos perceber pelo Gráfico 7.

Gráfico 7: Comparação entre Apache + PHP e Nginx + Node.js com foco em operações de CPU.



CPU	Média req / s	Média tempo / req
Apache + PHP	261.59	382 ms
Nginx + Node.js	1607.51	62 ms

Fonte: O Autor

3.4.3 Outros

Outros estudos foram desenvolvidos para avaliar e comparar o desempenho do Node.js com outras tecnologias de desenvolvimento Web. Sanchez (2015) faz uma

comparação teórica e prática entre a tecnologia estudada e PHP, utilizando o HHVM[63] para executar os códigos dessa linguagem. HHVM utiliza algumas mesmas técnicas de compilação que o V8, como JIT, e é atualmente utilizada e mantida pelo Facebook[64]. Ainda assim, nos testes de requisições simples realizados por Sanchez, Node.js conseguiu servir cerca de 17% a mais de requisições por segundo que o HHVM. Nos testes de processamento, para executar um algoritmo de ordenação, o Node.js usou cerca de 21% do tempo que o HHVM, e cerca de 75% de memória a menos.

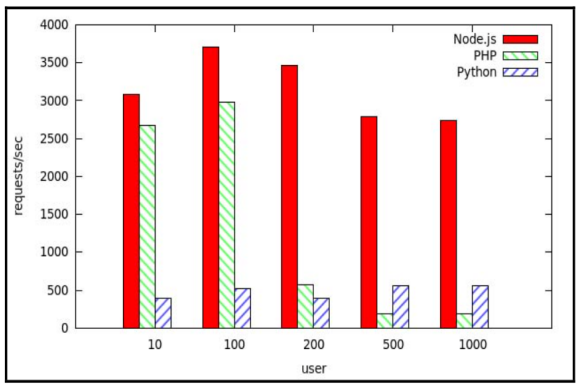
Segundo Lei, Ma e Tan (2014, p.1, tradução nossa):

> Para estudar e comparar o desempenho do Node.js, Python-Web e PHP, nós utilizamos testes de benchmarks e de cenários. Os resultados experimentais produziram dados de performance valiosos, mostrando que PHP e Python-Web aguentam muito menos requisições que o Node.js a partir de um determinado tempo. Em conclusão, nossos resultados mostraram claramente que Node.js é bastante leve e eficiente, o que é uma ideia que, para Websites de I/O intensivo, funciona melhor do que os outros dois […]

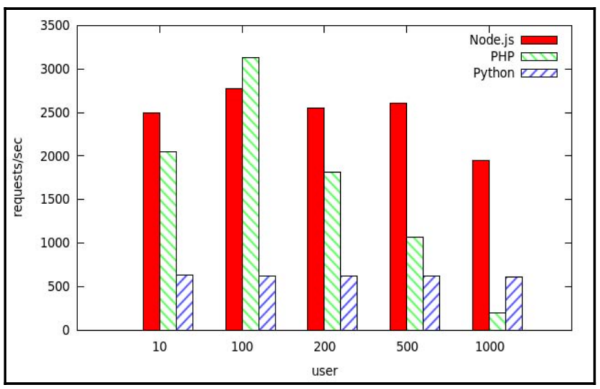
Eles fizeram dois testes: um para uma requisições simples, que responde “Hello world”, e outra que computa o 10º número de Fibonacci. Resumidamente, o desempenho do Node.js foi sempre muito mais rápido que Python-Web, e em 90% dos testes mais rápido que PHP, como mostram o Gráfico 8a e 8b.

Gráfico 8: Comparação de média de requisições por segundo para tarefas diferentes e número variável de requisições paralelas.

a) Responder com “Hello world”



b) Calcular Fibonacci(10)



Fonte: (LEI; MA; TAN, 2014)

[63] Máquina virtual *open source* projetada para executar programas escritos em PHP e Hack, que é uma variação de PHP.
[64] https://www.facebook.com/

4 PROJETO E IMPLEMENTAÇÃO

Começamos esse capítulo com a problemática que tentamos resolver com a aplicação desenvolvida, abordando alguns fatos e dados que sugerem que não há incentivo no Brasil para a participação de jovens em competições algorítmicas. Seguimos então com a implementação de uma plataforma brasileira de simulação em competições virtuais de programação utilizando a tecnologia Node.js. As seções seguintes discutem os aspectos funcionais e não-funcionais da aplicação, de forma a caracterizá-la pelo seu funcionamento e modo operacional. Terminamos este capítulo explicando detalhadamente a arquitetura desenvolvida para o sistema e as tecnologias envolvidas na concepção dele.

4.1 PROBLEMÁTICA

No ensino fundamental e médio existem várias olimpíadas que as escolas incentivam o aluno a participar, como: Olimpíada Brasileira de Matemática (OBM)[65], Olimpíada Brasileira de Física (OBF)[66] e Olimpíada Brasileira de Química (OBQ)[67], para citar algumas. Poucos sabem, porém, que existe a Olimpíada Brasileira de Informática (OBI) [68] , a Olimpíada Internacional de Informática (IOI, do inglês *International Olympiad in Informatics*) [69] , e a Competição Internacional de Programação para o Colegiado (ICPC, do inglês *International Collegiate Programming Contest*)[70], entre várias outras competições virtuais.

O mundo digital está tão em vigor hoje em dia que não é nada improvável que, muito em breve, jovens aprendam a programar na escola. Mas enquanto esse dia não chega, a Web é repleta de oportunidades de preparação para essas competições.

[65] http://www.obm.org.br/opencms/
[66] http://www.sbfisica.org.br/v1/olimpiada/2015/
[67] http://www.obquimica.org/
[68] http://olimpiada.ic.unicamp.br/
[69] http://www.ioinformatics.org/
[70] http://icpc.baylor.edu/

Enquanto que na Rússia e na China os alunos são informados desde cedo sobre várias oportunidades que fogem dos assuntos do ensino fundamental e médio, no Brasil não há incentivo algum para o estudante.

Oportunidades de aprendizado na Web não faltam. O Codeforces[71], plataforma russa, mas com idioma disponível em inglês, promove competições regulares de algoritmos e sempre disponibiliza um editorial com a explicação da resolução dos problemas. O TopCoder [72] é outro exemplo de plataforma de competições relacionadas à computação que também ensina, e ainda paga os melhores competidores. Essas competições fornecem um conjunto de questões, e o participante deve escrever um programa de computador em uma linguagem de sua preferência (geralmente C, C++ ou Java) que resolva esse problema. Enquanto há cerca de 4000 usuários da Rússia e 4500 da China cadastrados no Codeforces, esse número chega a menos de 400 para usuários do Brasil[73].

4.2 OBJETIVO

Este trabalho envolve a implementação de uma plataforma brasileira de competições em problemas algorítmicos, entitulada Maratonando, com o objetivo de disseminar a ideia de programação competitiva no Brasil e alavancar a prática de treinamento para as competições reais, como OBI, IOI e ICPC, por meio de uma ferramenta de simulação de competições virtuais. O ambiente fornece uma oportunidade de criar suas próprias competições e ainda participar de competições criadas por outros usuários que buscam se aperfeiçoar e trocar conhecimento.

As diversas regras que são atribuídas às competições reais, como o congelamento do placar e os tipos de competidores permitidos (times ou indivíduos) são incluídas na plataforma construída, de forma que o competidor possa se preparar nos moldes das reais competições. No mais, é importante citar que no momento não há nenhum sistema desenvolvido conhecido que forneça esse método de treino.

---

[71] http://codeforces.com/
[72] http://www.topcoder.com/
[73] Dado extraído em junho de 2015 de http://codeforces.com/ratings.

4.3 ASPECTOS FUNCIONAIS

Os aspectos funcionais são utilizados para descrever as funções do sistema, ao invés de descrever a forma como ele opera. A seguir, descrevemos as principais características da plataforma no nível funcional, não levando em conta ainda detalhes de desempenho e usabilidade da plataforma.

a) **Integração com OJs –** a plataforma é integrada aos principais OJs (*Online Judges*)[74]: UVa[75], LiveArchive[76], Timus[77], Spoj[78], SpojBr[79] e URI[80]. Dessa forma, qualquer usuário pode criar uma competição virtual com questões de qualquer um desses OJ, pois todas essas questões são importadas para o sistema local, de forma que quando um usuário escolhe os problemas da competição, ele tem acesso apenas à lista de opções válidas.

b) **Tipos de perfis –** a plataforma possui dois tipos de perfis: usuários individuais e times. O perfil de usuário individual é criado a partir do fluxo de cadastro no sistema — nome, email, login e senha. O perfil de time é criado a partir de uma composição de perfis individuais. As competições podem ser configuradas para aceitar apenas perfis individuais, apenas times ou ambos. Um usuário pode escolher o time que deseja representar, dentre a lista de times do qual faz parte, na hora de se inscrever em competições de times.

c) **Tipos de competições –** as competições criadas na plataforma podem ser configuradas em diversos níveis, como acessibilidade (particular ou pública), tempo de duração, *frozen*[81] e *blind*[82] e tipo de competidores (individuais, times ou ambos). No caso de ser particular, o criador ainda define uma senha para permitir apenas determinados usuários de participar da competição.

---

[74] É chamado Online Judge o sistema online que compila e dá o veredito de submissões para questões do sistema.
[75] https://uva.onlinejudge.org/
[76] https://icpcarchive.ecs.baylor.edu/
[77] http://acm.timus.ru/
[78] http://www.spoj.com/
[79] http://www.spoj.com/
[80] https://www.urionlinejudge.com.br/
[81] Período da competição nos moldes da ICPC no qual o placar fica congelado, e os usuários só tem acesso ao resultado de suas próprias submissões.
[82] Parecido com o *frozen*, mas o usuário não tem acesso ao resultado de nenhuma submissão.

d) **Módulos auxiliaries –** dois módulos foram criados em Node.js: um que envia os códigos submetidos (*Submitter*) no sistema para os respectivos OJ, e outro que coordena a verificação do veredito dessas submissões (*Judger*). Esses módulos provam a flexibilidade do Node.js de não ser apenas uma plataforma para desenvolvimento Web, mas também para criação de serviços que executam determinadas tarefas.

e) **Classificação dos problemas –** durante a competição, toda vez que um indivíduo ou time acertar uma questão, ele pode classificá-lo por assunto e dificuldade. Isso abre inúmeras possibilidades para futuras versões, como prover um filtro de questões que ajude em treinos nivelados ou concentrados em determinados assuntos, ou um gerador aleatório de lista de problemas para serem resolvidos.

4.4 ASPECTOS NÃO-FUNCIONAIS

Aspectos não-funcionais relatam a forma como o sistema opera, especificando critérios como desempenho, segurança e tecnologias envolvidas. De forma geral, os aspectos não-funcionais não afetam diretamente na forma como um usuário interage com o sistema, pois o produto visível e manuseável para o cliente depende apenas dos aspectos funcionais; pode, contudo, afetar indiretamente, pois condições de eficiência e usabilidade são tão importantes quanto a funcionalidade da aplicação.

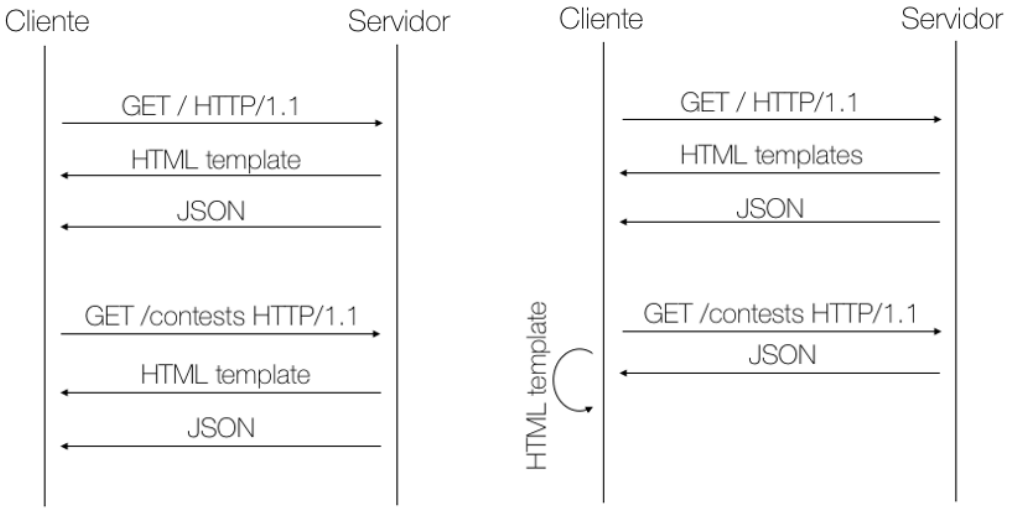
a) **Desempenho –** empregamos algumas técnicas para agilizar o tempo de resposta das requisições e diminuir os recursos de CPU utilizados no servidor. A primeira técnica é a *minificação* e compressão de recursos de JavaScript e CSS[83], que consiste em renomear variáveis e comprimir os arquivos em um só. Aplicando essa técnica, reduzimos em 27% o tamanho do CSS e 31% o tamanho do JavaScript fornecido pelo nosso servidor. Outra técnica empregada é delegar a computação dos recursos para o cliente, de forma que o usuário recebe o template HTML e os dados, e seja responsável

[83] *Cascading Style Sheets* é uma linguagem utilizada para definir a apresentação de documentos escritos em uma linguagem de marcação.

por renderizar a página localmente — *Client-Side rendering*. Outra técnica de aumento de desempenho empregada é *template caching*, que será explicada no tópico de usabilidade por ser extremamente relacionada à usabilidade do sistema. Por fim, para servir arquivos estáticos de forma rápida, utilizamos o servidor de *reverse proxy* Nginx, comentado extensivamente neste trabalho.

b) **Usabilidade –** a aplicação desenvolvida é uma Single-Page Application[84] (SPA), para dar ao usuário uma experiência de fluidez entre as páginas visitadas. Todos os recursos necessários para gerar outras páginas são obtidos assincronamente do servidor, sem que um novo carregamento de página se faça necessário. Para o HTML, pré-armazenamos todos os *templates* no cliente por meio de uma técnica denominada *Template Caching*. Por mais que isso aumente o tamanho da resposta da primeira requisição ao servidor, evita futuros ciclos de requisição de HTML, pois os *templates* já serão salvos no cliente. As próximas requisições são feitas por AJAX ao servidor apenas para recuperação de dados. O diagrama de sequência simplificado de *Template Caching* é mostrado na Figura 14. Percebemos que quando o cliente já tem o *template* salvo no seu lado, apenas uma requisição é necessária.

Figura 14: Modelo de requisição cliente-servidor com apenas Client-Side rendering (esq.) e SPA + Template Caching + Client-Side rendering (dir.).



Fonte: O Autor

[84] Aplicações que aparentam funcionar em uma única página, sem necessidade de recarregamento.

4.5 ARQUITETURA

A pilha de desenvolvimento Web envolve várias camadas, como a LAMP, vista no Capítulo 1, a qual define o sistema operacional, o servidor Web, o sistema de banco de dados e a linguagem de programação. Para o sistema desenvolvido, utilizamos a MEAN *stack*[85]: MongoDB, Express.js[86], AngularJS[87] e Node.js. Essa pilha de desenvolvimento define desde a plataforma de desenvolvimento utilizada até a tecnologia de banco de dados. Dizemos que essa pilha de desenvolvimento é *full-stack* JavaScript pois todas as camadas lidam naturalmente com a mesma linguagem. Os documentos são armazenados em formato JSON no MongoDB, cada *query* é construída no mesmo formato em Express.js e Node.js e, eventualmente, o lado do cliente, por meio do AngularJS, lida com o mesmo tipo de objeto. Depuração e administração do banco de dados se torna muito mais fácil quando os objetos que são armazenados no banco de dados são os mesmos objetos que todo o resto da sua pilha de desenvolvimento enxerga.

Na MEAN *stack*, cada ferramenta é de fundamental importância para o funcionamento do sistema. O MongoDB é um banco de dados NoSQL[88] orientado a documentos. Os documentos são inseridos e extraídos em formato JSON, no mesmo modelo seguido para todo o resto da pilha de desenvolvimento, de forma que nenhuma conversão se faz necessária. O módulo utilizado do NPM que conecta o Node.js ao MongoDB, faz a modelagem dos dados, lida com a validação dos tipos e facilita as operações de CRUD[89] é o Mongoose[90].

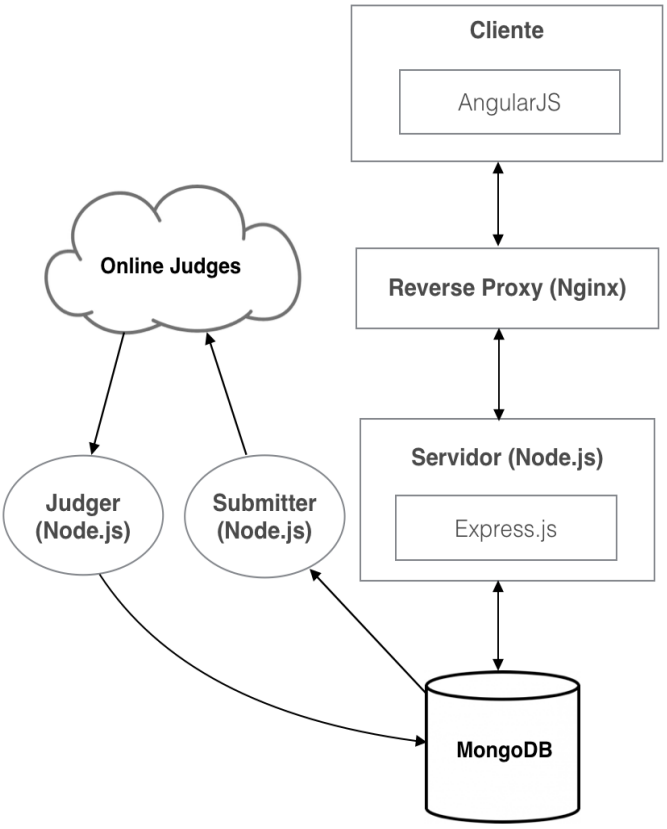
O Express.js é o *framework* Web que executa sobre a plataforma Node.js. Tudo o que fazemos com o Express.js, conseguimos fazer apenas com o Node.js, mas de uma forma muito mais trabalhosa. Essa ferramenta provê um conjunto minimalista de funções para desenvolvimento Web. Em outras palavras, o Express.js é para Node.js o que o Ruby on Rails é para Ruby, ou o Django é para Python. Express.js facilita todas as tarefas de um servidor Web: desde o gerenciamento de rotas à manipulação

[85] http://mean.io/
[86] http://expressjs.com/
[87] https://angularjs.org/
[88] Termo utilizado para definir uma classe de banco de dados não-relacionais.
[89] Create, Read, Update e Delete são as quatro operações básicas de um banco de dados persistente.
[90] http://mongoosejs.com/

de requisições e respostas HTTP. Nessa camada, são definidas todas as rotas não-estáticas da aplicação — as rotas estáticas estão em um diretório /public que é servido pelo Nginx. Como a aplicação foi desenvolvida sob a técnica de *Client-Side rendering*, o roteamento implementa a arquitetura REST[91], de forma que todas as requisições AJAX do cliente para o servidor são para rotas definidas na API REST .

O AngularJS funciona no lado do cliente e torna fácil tanto a criação de uma *Single-Page Application* como a aplicação da técnica de *Client-Side rendering*. AngularJS emprega o conceito de *Model-View-Controller*[92] (MVC), de forma que os modelos são adquiridos por meio da API REST, definida no servidor. A renderização das telas fica a cargo dos controladores, que estão no cliente, e agem em conjunto

Figura 15: Arquitetura do Maratonando. Uma seta direcionada de uma entidade X para uma entidade Y significa que algum recurso de X é enviado para Y.



Fonte: O Autor

[91] Representational State Transfer é utilizado para requisição e resposta por HTTP de recursos do servidor, seja um pedido de criação, deleção, atualização ou recuperação.
[92] Padrão de engenharia de software que separa as representações internas de informação da interação do usuário com elas.

com os modelos e os *templates* para gerar a tela com dados para o usuário, minimizando o processamento feito no servidor. Com o AngularJS, fica muito mais simples associar elementos do HTML a modelo de dados providos do servidor.

A Figura 15 mostra a arquitetura do projeto em termos dos módulos e tecnologias envolvidas. É importante perceber os módulos *Submitter* e *Judger* também foram implementados em Node.js, mas não fazem parte do servidor Web diretamente. Esses dois módulos são serviços que executam em plano de fundo e fazem a correção das questões a partir de servidores externos. O *Submitter* lê do banco de dados as submissões pendentes, que ainda não foram corrigidas, e as envia para os respectivos OJ. Uma vez submetidas, é responsabilidade do módulo *Judger* verificar os resultados das submissões e disponibilizá-los no banco de dados.

4.6 RESULTADOS

Nas versões iniciais do projeto, os módulos de *Judger* e *Submitter* foram implementados junto ao servidor Web. Com isso, uma boa parte do processamento do servidor era gasto iterando sobre a lista de requisições pendentes e submetendo-as para os OJ, além de sempre verificar se as correções já tinham sido feitas. Dessa forma, o processamento de muitas requisições ficava bloqueado até que esses módulos terminassem sua execução, o que afetava diretamente a experiência do usuário com o sistema, que esperava muitos segundos para ter a página carregada. Quando separamos essas tarefas em módulos distintos, conseguimos vários usuários utilizando o sistema sem mais nenhum problema, obtendo uma média de 180 ms por requisição.

Durante a implementação do sistema, utilizamos um módulo do NPM para automatizar tarefas — GruntJS[93]. Visto que todo o JavaScript e CSS do cliente era *minificado*, e os templates HTML eram inseridos no JavaScript para podermos aplicar a técnica de *Template Caching,* era necessário fazer esse processo toda vez que o sistema fosse ser testado. O GruntJS foi muito útil nesse ponto, pois ele observava todos os arquivos relativos ao cliente do projeto e, quando eles eram modificados, o

[93] http://gruntjs.com

GruntJS executava a minificação do JavaScript e CSS e a mudança do HTML em plano de fundo.

O uso do Nginx também provou melhorar muito o desempenho da aplicação. Para os recursos estáticos testados, o Nginx retornava a resposta cerca de 5 vezes mais rápido que o Node.js. Isso também sugere que no futuro podemos transformar algumas páginas com dados dinâmicos em estáticos, como competições passadas. Atualmente, os dados referentes a elas são recuperados do banco de dados e passados para serem renderizados no cliente. Porém, a partir do momento que uma competição termina, sabemos que ela não pode mais mudar, e podemos transformá-la numa página estática para que o Nginx passe a servi-la mais rapidamente.

Os testes de interação do usuário com o sistema foram feitos com alunos de um grupo de estudo para competições algorítmicas do Centro de Informática, na Universidade Federal de Pernambuco. Diversas competições foram simuladas e funcionaram muito bem na plataforma. As maiores dificuldades ocorreram quando algum OJ utilizado numa determinada competição ficava fora do ar. Nesse caso, nenhum feedback era entregue aos usuários, que ficavam por muito tempo sem ver o resultado de suas submissões. Essas e outras sugestões foram planejadas para serem implementadas futuramente, e foram incluídas na seção de trabalhos futuros.

5 CONCLUSÕES

O estudo conduzido nessa monografia investigou uma poderosa plataforma de desenvolvimento Web, assim como os componentes que a sustentam. Foi possível entender as técnicas subjacentes que são aplicadas para alavancar o desempenho do sistema, e aplicamos na prática o conhecimento obtido, desenvolvendo uma aplicação original em Node.js. O estudo detalhado da tecnologia envolveu não só sua arquitetura e componentes, mas também seu desempenho em comparação a outros mecanismos de desenvolvimento Web. As avaliações de desempenho foram divididas em duas partes: primeiro, medimos a velocidade de execução de algoritmos por parte do JavaScript V8 e outras linguagens de programação; segundo, medimos o desempenho do Node.js em termos de requisições por segundo e tempo médio por requisição. Tanto o V8 quanto o Node.js — que é intrinsecamente ligado ao primeiro — se sobressaíram em todos os testes realizados, provando a eficiência da tecnologia estudada.

Consideramos que a popularização e o crescimento da comunidade de desenvolvedores de SSJS nos últimos anos fortifica o argumento de que essa tecnologia será cada vez mais adotada em projetos. Vários trabalhos e pesquisas estão sendo desenvolvidos academicamente sobre desenvolvimento de aplicações Node.js, e várias grandes corporações já migraram para essa tecnologia ou desenvolvem projetos baseados nela[94]. Tilkov e Vinoski (2010) fizeram um estudo de desenvolvimento de aplicações de rede com alto desempenho em Node.js, envolvendo os conceitos de programação concorrente e orientação a eventos, de forma a maximizar o desempenho da plataforma em contextos *CPU bound*. Stecca et al. (2013) investigam a utilização eficiente de recursos do sistema, como *threads* e memória, utilizando I/O assíncrono.

Outro aspecto bastante relevante de qualquer plataforma de desenvolvimento Web é segurança. Ojamaa e Düüna (2012) fazem um estudo avaliativo da segurança do Node.js, apontando algumas possíveis vulnerabilidades que o desenvolvedor deve se atentar. Além disso, o artigo aponta duas fragilidades da plataforma, e dá

[94] O Google dispõe de uma equipe para manter e evoluir o AngularJS. O Netflix, PayPal e LinkedIn atualmente já migraram suas plataformas para Node.js.

recomendações para construir e configurar aplicações Web seguras e resilientes. Como esse trabalho não abordou os aspectos de segurança da plataforma, recomendamos fortemente a leitura desse artigo.

Por fim, a aplicação desenvolvida foi importante não só no contexto de estudo para este trabalho, mas também no importante espectro que assume: plataforma brasileira para treinos em competições algorítmicas. Aplicamos na prática todos os conceitos de otimização trabalhados aqui para prover um sistema dinâmico, escalável e de mínima latência, que é o que estamos preocupados desde o início. Acreditamos ter conseguido alcançar o objetivo de implementar um sistema eficiente e útil para muitas instituições de ensino, que buscam uma forma de sistematizar os treinos de programação competitiva.

5.1 TRABALHOS FUTUROS

A fim de melhorar a experiência do usuário com o sistema, elaboramos uma série de melhorias de funcionalidade que devem ser implementadas. Não obstante, visamos outras ações que não envolvem somente novas funções, mas também a divulgação ampla da plataforma para instituições de ensino e testes de segurança e vulnerabilidade da aplicação. A seguir, descrevemos algumas mudanças planejadas, funcionais, não-funcionais e estratégicas:

a) **Status dos OJ –** Os servidores dos *Online Judges* integrados à aplicação podem ficar *offline*, de forma que as submissões feitas ficam sem resultados por muito tempo, e os usuários não recebem feedback nenhum sobre isso. Criaremos um módulo verificador para esses servidores externos, de forma que quando eles saiam do ar, tenhamos acesso a essa informação o mais rápido possível para disponibilizar publicamente.

b) **Corretor local –** outro importante ponto é transformar a aplicação desenvolvida em um OJ próprio, de forma que possamos realizar correções locais de questões. A implementação não é trivial, pois os códigos serão compilados e rodados localmente e podem ser maliciosos. No entanto, esse passo garante mais liberdade e autonomia da plataforma, de forma que muitas questões

possam ser migradas com o tempo para o corretor local, diminuindo a dependência com servidores externos.

c) **Divulgação –** uma decisão futura de produto será divulgar a plataforma para as páginas oficiais das organizações que incentivam e divulgam competições de programação, como a Maratona de Programação[95]. Muitas universidades buscam um modelo de treinamento automatizado, mas não há nenhuma ferramenta em vigor na Internet atualmente que sistematize competições virtuais simuladas de programação. Dessa forma, esperamos que a existência da aplicação desenvolvida se torne conhecimento público de todas as instituições que preparam seus alunos para essas competições.

d) **Testes de segurança –** junto aos testes unitários e de integração, que visam manter a integridade e consistência da plataforma, precisamos realizar testes de segurança e vulnerabilidade para evitar ataques de usuários maliciosos. Essas questões não envolvem necessariamente uma invasão ao banco de dados ou ao sistema que executa a aplicação, mas podem também ser uma falha no sistema que faz o servidor sair do ar. Qualquer tipo de ação executada pelo servidor que não foi prevista pelo desenvolvedor deve ser investigada.

e) **Open source –** esse passo é crucial para que outras pessoas possam estudar a plataforma desenvolvida e aplicar o conhecimento em suas instituições, além de ajudar-nos a descobrir os erros na aplicação. Tornar um produto *open source* é uma decisão estratégica que muitas empresas deixam de tomar por razões monetárias e de competitividade do mercado. Porém, o objetivo da aplicação desenvolvida nunca foi lucrativo, mas apenas educativo, e torná-lo *open source* não seria prejudicial aos planos que temos para essa plataforma.

[95] http://maratona.ime.usp.br/

REFERÊNCIAS

BERNERS-LEE, Tim, “The Original HTTP as defined in 1991”. 1991. Disponível em: <http://www.w3.org/Protocols/HTTP/AsImplemented.html> Acesso em: Julho 2015.

BERNERS-LEE, Tim; FIELDING, Roy; FRYSTYK, Henrik; GETTYS, Jim; MOGUL, Jeffrey, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, DOI 10.17487/RFC2068, Jan. 1997.

BONVIN, Nicolas. Serving static files: a comparison between Apache, Nginx, Varnish and G-WAN. May 2011. Disponível em: <https://nbonvin.word-press.com/2011/03/14/apache-vs-nginx-vs-varnish-vs-gwan/> Acesso em: Maio 2015.

CHAMBERS, Craig; LEE, Elgin; UNGAR, David, An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. OOSPLA ’89 Conference Proceedings, pp. 49-70. 1989.

CONROD, Jay, A tour of V8: Crankshaft, the optimizing compiler. April 2013. Disponível em: <http://jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler> Acesso em: Maio 2015.

CROCKFORD, Douglas, “JSON: The Fat-Free Alternative to XML”. XML Conference, 2006, Boston.

CROCKFORD, Douglas. JavaScript: The Good Parts. 1 ed. California: Yahoo! Press, Maio 2008. 172 p.

D., Remi. A little holiday present: 10,000 reqs/sec with Nginx!, WebFaction. Dez. 2008. Disponível em: <http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/> Acesso em: Maio 2015.

DEUTSCH, L. Peter; SCHIFFMAN, Allan M., Efficient implementation of the smalltalk-80 system, Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 297-302, Jan. 15-18, 1984.

ELLINGWOOD, Justin. Apache vs Nginx: Practical Considerations, Digital Ocean Inc. Jan 2015. Disponível em: <https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations> Acesso em: Maio 2015.

FORTUNA, Emily; ANDERSON, Owen; CEZE, Luis; EGGERS, Susan, "A limit study of JavaScript parallelism," Workload Characterization (IISWC), 2010 IEEE International Symposium on , vol., no., pp.1,10, 2-4

GARRET, Jesse J., Ajax: A New Approach to Web Applications. AdaptivePath.com, Fev. 2005. Disponível em: <https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php> Acesso em: Maio 2015.

GU, Xiao-Feng; YANG, Le; WU, Shaoquan, "A real-time stream system based on node.js," Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on , vol., no., pp. 479,482, 19-21 Dez. 2014

KEGEL, Dan. The C10K problem. Disponível em: <http://www.kegel.com/c10k.html/> Acesso em: Abril 2015.

LEI, Kai; MA, Yinning; TAN, Zhi, "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js", Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on, pp. 661 - 668

MCCARTHY, John, "Recursive functions of symbolic expressions and their computation by machine, Part I". Communications of the ACM 3 (4): 184–195. April 1960.

MARDAM, Azat. Practical Node.js: Building Real-World Scalable Web Apps. Apress. 2012.

OGASAWARA, Takeshi, "Workload characterization of server-side JavaScript," Workload Characterization (IISWC), 2014 IEEE International Symposium on , vol., no., pp.13,21, 26-28 Out. 2014

OJAMAA, Andres; DUUNA, Karl, "Assessing the security of Node.js platform," Internet Technology And Secured Transactions, 2012 International Conference for , vol., no., pp.348,355, 10-12 Dez. 2012

POSTEL, Jonathan, “Simple Mail Transfer Protocol”, RFC 821, DOI 10.17487/RFC0821, Ago. 1982.

POSTEL, Jonathan, “Transmission Control Protocol”, RFC 793, DOI 10.17487/RFC0793, Set. 1981.

POSTEL, Jonathan, “User Datagram Protocol”, RFC 768, DOI 10.17487/RFC0768, Ago. 1980.

POSTEL, Jonathan; REYNOLDS, Joyce, “File Transfer Protocol (FTP)”, RFC 959, DOI 10.17487/RFC0959, Out. 1985.

RAPPL, Florian, Operation Performance Evaluation. Code Project. Jan. 2012. Disponível em: <http://www.codeproject.com/Articles/304935/Operation-Performance-Evaluation> Acesso em: Maio 2015.

SANCHEZ, Roberto. “Comparing Node.js vs PHP Performance”, HostingAdvice. Abril 2015. Disponível em: < http://www.hostingadvice.com/blog/comparing-node-js-vs-php-performance/> Acesso em: Jun. 2015.

STECCA, Michele; FORNASA, Martino; BAGLIETTO, Pierpaolo; MARESCA, Massimo, "Scalable Service Composition Execution through Asynchronous I/O", Services Computing (SCC), 2013 IEEE International Conference on, pp. 312 - 319

THE COMPUTER LANGUAGE BENCHMARKS GAME. Disponível em: <http://benchmarksgame.alioth.debian.org/> Acesso em: Maio 2015.

THE ECMASCRIPT LANGUAGE SPECIFICATION, Standard ECMA-262, ECMA International, ed. 3. Dez. 1999.

TIWARI, Devesh; SOLIHIN, Yan, "Architectural characterization and similarity analysis of sunspider and Google's V8 Javascript benchmarks," Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on , vol., no., pp.221,232, 1-3 Abril 2012

TILKOV, Stefan; VINOSKI, Steve, "Node.js: Using JavaScript to Build High-Performance Network Programs," Internet Computing, IEEE, vol.14, no.6, pp.80,83, Nov.-Dez. 2010

WILSON, Chris. Performance Tips for JavaScript in V8. Disponível em: <http://www.html5rocks.com/en/tutorials/speed/v8/> Acesso em: Abril 2015.