Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Estudo comparativo entre ferramentas de automação via RPA

Paulo Diógenes Silva Salgado Filho

Trabalho de Graduação

Recife
18 de Maio de 2022

Universidade Federal de Pernambuco
Centro de Informática

Paulo Diógenes Silva Salgado Filho

Estudo comparativo entre ferramentas de automação via RPA

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Prof. Dr. Carlos André Guimarães Ferraz*

Recife
18 de Maio de 2022

Agradecimentos

Agradeço a Deus primeiramente por permitir a realização deste trabalho que conclui uma importante fase da vida, bem como por ter colocado pessoas tão especiais em minha jornada pela graduação.

Agradeço ao professor Carlos Ferraz por ter me aceitado como orientando, compartilhado seu conhecimento e disponibilizado seu tempo ao longo dos últimos meses, foi um verdadeiro privilégio ter tal oportunidade.

Agradeço à equipe do CIN-UFPE, com seus professores e colaboradores que sempre se mostraram empáticos e acolhedores.

Agradeço especialmente aos meus pais, Paulo Salgado e Ivaneide Félix e também a toda minha família por ter sempre promovido todo o suporte necessário para que eu pudesse continuar seguindo em frente, e por ensinar a sempre batalhar dando o meu melhor.

Agradeço aos muitos amigos que pude adquirir durante os anos de graduação em especial: Anderson, Daniel e Matheus a quem o tempo levou a caminhos diferentes mas sem nunca se afastar, vocês tornaram a jornada muito mais prazerosa de seguir.

Agradeço a Victor Melia o qual durante vários momentos da graduação se mostrou não só um grande amigo mas uma inspiração pessoal.

Agradeço também à toda equipe da Beyond Co., especialmente a Victor Aurélio, Nathan Freire e Levi Nóbrega que têm sido parte especial da minha vida.

Por fim, agradeço a muitos outros que tiveram contribuição para a realização deste trabalho mas que não pude citar com palavras, a todos vocês meu muito obrigado!

Resumo

O mundo vive um processo de digitalização sem precedentes, soluções digitais para problemas do dia a dia não são novidades, mas foram potencializadas de forma quase impositiva devido à pandemia. Atualmente, processos que exigem presença física estão cada vez mais decadentes: Processos jurídicos, consultas médicas e até mesmo boa parte dos trabalhos tradicionais hoje em dia possuem algum correlato digital, e assim como funciona no meio físico, no mundo digital existem várias situações de trabalho repetitivo que podem ser automatizados através do uso de robôs. Atualmente existem várias ferramentas que permitem automatizar fluxos de trabalho na internet, verdadeiras empresas são construídas ao redor dessas soluções. Duas destas ferramentas são as bibliotecas: *Puppeteer.Js* e *Selenium*. O objetivo deste trabalho é fornecer um estudo comparativo entre estas duas tecnologias, apontando pontos positivos e negativos e indicando em que situações poderá haver vantagem para alguma delas.

Palavras-chave: RPA, automação, *Puppeteer.js*, *Selenium*, robô.

Abstract

The world is experiencing an unprecedented process of digitization, digital solutions to everyday problems are not new, but they were leveraged in an almost imposing way due to the pandemic. Currently, processes that require physical presence are increasingly decadent: Legal processes, medical consultations and even a good part of the traditional office work today have some digital correlate, and just as it works in the physical, in the digital world there are several situations of repetitive work that can be automated through the use of robots. Currently, there are several tools that allow automating workflows on the internet, real companies are built around these solutions. Two of these tools are the libraries: Puppeteer.js and Selenium. The objective of this work is to provide a study comparison between these two technologies, pointing out positive and negative points and showing which situations there could be an advantage for any of them.

Keywords: RPA, automation, *Puppeteer.js*, *Selenium*, bot.

Lista de Figuras

Lista de Algoritmos

Sumário

CAPÍTULO 1

Introdução

1.1 Contexto

A computação tem se tornado cada vez mais presente no cotidiano das empresas. Atualmente, são raros os casos de indústrias ou comércios que funcionem sem um auxílio do mundo digital, seja através de websites, software para controle de recursos, gerenciamento de pessoas, meios de pagamentos. A lista é enorme e a tendência é de crescimento. Uma outra forma, não tão conhecida, mas extremamente útil para as empresas é a automação de processos, ou como é mais conhecida, Automação Robótica de Processos (do inglês, RPA – *Robotic Process Automation*). Como o nome sugere, a prática busca automatizar um procedimento, antes feito manualmente, através do emprego de robôs.

O termo RPA é usado para designar softwares que executam tarefas feitas anteriormente de forma manual [Kaya, Turkyilmaz e Birol 2019], isto é, os robôs são programados para serem capazes de usar a mesma interface que seres humanos, simulando as ações de um operador do sistema, eliminando assim a necessidade de desenvolvimento de *APIs* ou da realização de grandes mudanças nos sistemas já utilizados. Normalmente o principal objetivo da aplicação da automação de processos é eliminar trabalhos repetitivos e padronizados, muito comuns em escritórios [Willcocks, Lacity e Craig 2015].

O mercado de RPA já movimenta bilhões de dólares pelo mundo [Anagnoste 2017] devido a seus ganhos comprovados em produtividade, padronização, disponibilidade, escalabilidade e flexibilidade [Meironke e Kuehnel 2022], que garantem um retorno rápido do investimento e aumentam a satisfação dos trabalhadores com seu trabalho [Willcocks, Lacity e Craig 2015]. Durante a pandemia de COVID-19, situação que obrigou empresas a fecharem suas portas e trabalhadores a trabalharem de suas casas, a adoção da automação permitiu manter a continuidade dos processos de negócios e em alguns casos até aumentando mais ainda a produtividade [Siderska 2021].

Atualmente existem dezenas de software voltados para a automação de processos. Este trabalho busca comparar, de forma quantitativa, avaliando o consumo de recursos de CPU e memória, duas bibliotecas de código aberto (*open-source*) voltadas para a automação web, isto é, automação de processos executados em um navegador, tais quais: preenchimento de formulários, extração de dados de páginas da web, teste de websites, etc. São elas: *Selenium* [Selenium 2022] e *Puppeteer.Js* [LLC 2022].

1.2 Objetivos

1.2.1 Objetivo geral

Realizar estudo comparativo entre as bibliotecas Puppeteer.Js e Selenium (Python) no contexto da automatização de fluxos de trabalho na web.

1.2.2 Objetivos específicos

- Entender fluxos automatizáveis de trabalho e em que situações podemos aplicar automação;
- Entender e aplicar métodos de comparação de performance entre aplicações.
- Entender como construir contêineres e monitorar sua performance.
- Avaliar em que situações possuímos alguma vantagem na escolha de uma determinada biblioteca em favor de outra.

CAPÍTULO 2

Metodologia

Neste capítulo, os algoritmos, ferramentas e bibliotecas utilizados para as comparações serão apresentados e contextualizados, alguns conceitos gerais também serão discutidos.

2.1 Visão geral

Para a realização da comparação entre as ferramentas [Persson 2019], os robôs foram desenvolvidos usando cada uma das duas bibliotecas, de modo que cada um deles realize as mesmas tarefas, seguindo os mesmos passos, na mesma ordem e com os mesmos parâmetros. Foram considerados dois cenários neste experimento com as principais aplicações de RPA: (1) a extração de dados da web e (2) a automação de preenchimento de formulários. De modo geral, é possível dividir a atuação dos robôs nas fases apresentadas na figura 2.1

As fases do algoritmo são descritas como segue:

- Entrada de parâmetros: entrada de dados relevantes para a automação, como URLs e parâmetros a serem inseridos.
- Invocação do navegador: etapa onde o navegador é iniciado e os parâmetros de navegação como tamanho da tela, uso de memória e modo de navegação são definidos.
- Aguardar o carregamento: é feita a espera do carregamento da página para prosseguir a automação.
- Execução do processo automatizado: é nessa fase onde é executada a automação do processo, isto é, o principal objetivo a ser alcançado.
- Compilação dos resultados: última fase da automação, é onde os resultados da fase anterior são compilados. Ao final, pode-se ter um comprovante do trabalho feito, um relatório de testes ou simplesmente os dados recuperados em um *scraping*.

As rotinas desenvolvidas foram inseridas em contêineres Linux, ambientes completamente dissociados entre si. Cada um dos robôs desenvolvidos foi inserido em um contêiner apropriado, contendo apenas o mínimo necessário para a execução do processo. Sendo assim, pôde-se fazer um monitoramento mais apropriado com relação ao consumo de CPU, consumo de memória e I/O de rede.

Cada rotina desenvolvida foi executada 10 vezes, e os dados de consumo de recurso foram colhidos usando a ferramenta de leitura nativa do *docker*. Após o experimento, os dados de

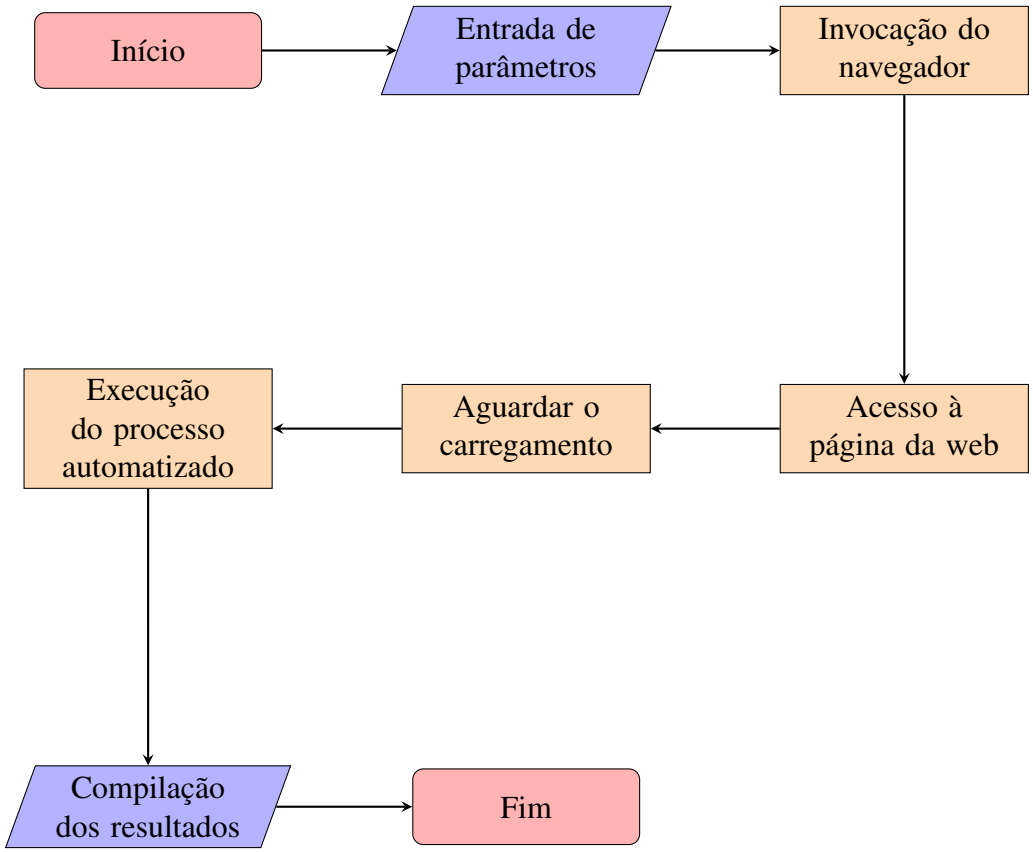

Figura 2.1: Fluxograma ilustrando as etapas globais executadas durante um trabalho de automação RPA. Fonte: adaptado de [Hindel, Cabrera e Stierle 2020].

consumo coletados passaram por limpeza e remoção de *outliers* causados por eventuais atrasos de pacotes. Foi então feito um estudo do consumo médio de CPU (em %) e consumo médio de memória (em Mib) para cada uma das execuções. Mediana, variância e desvio padrão também foram levadas em consideração para a confiabilidade dos dados apurados.

2.2 Conceitos preliminares

2.2.1 Selenium

Selenium é atualmente uma das mais usadas ferramentas para automação de processos na web, apesar de não ter sido criada originalmente para este propósito, mas sim para testes de software. Seu funcionamento se dá a partir da automação do navegador via um software de integração chamado *webdriver*, o webdriver é responsável por traduzir os comandos escritos em código para rotinas interpretadas pelo navegador. Outra notável característica de Selenium é o fato de existirem bibliotecas em diversas linguagens de programação, para este trabalho usaremos a versão em *Python* por ser a linguagem interpretada mais usada por quem usa Selenium [García et al. 2020].

O Selenium por si só não possui um navegador, sendo necessário que o ambiente de execução forneça um navegador, bem como o *webdriver* compatível com o mesmo.

2.2.2 Puppeteer

Puppeteer, ao contrário do Selenium, surgiu com propósitos gerais de automatizar o que quer que seja feito através de um navegador, como: testar aplicações, gerar capturas de tela ou imprimir, *web scraping*, *server side rendering* ou simplesmente automatizar tarefas feitas manualmente. Puppeteer.Js já vem com uma instância do navegador *Chromium* em seus arquivos, não sendo necessário realizar a instalação de softwares adicionais para seu uso. Sua interação com o navegador se dá a partir do *DevTools Protocol*, um protocolo de comunicação criado pela Google [Inc 2020] que permite interação direta com o navegador.

2.2.3 Contêineres

Contêineres são estruturas que permitem isolar o ambiente de execução de uma aplicação do ambiente da máquina que a hospeda. Um contêiner contém, além do programa a ser executado, todas as dependências e arquivos necessários para o funcionamento, facilitando assim a execução em diferentes ambientes, migração e escalabilidade. Por estar em um ambiente totalmente à parte do sistema hospedeiro, temos mais facilidade em medir o consumo real dos recursos usados pelo programa em execução. Uma das mais conhecidas implementações de contêiner é o *Docker*[1]

2.3 Etapas dos algoritmos

Nesta seção veremos as configurações iniciais e o passo a passo executado por cada um dos robôs avaliados neste experimento. Para uma construção mais linear da narrativa do experimento, primeiramente será mostrada a configuração do ambiente, e por fim os algoritmos usados.

2.3.1 Configurações iniciais – Selenium

Para a execução do robô em *Selenium* foi criado um contêiner usando uma imagem *python 3.8* como base, que é a versão mais recomendada para uso no momento em que o experimento foi executado. Em seguida, o ambiente é atualizado para receber os pacotes mais recentes para o ambiente e então é feita a instalação do navegador *Google Chrome* e do *webdriver* compatível. Por fim, copiamos todo o repositório para o ambiente do contêiner e instalamos, usando o gerenciador de pacotes *PIP*, as dependências de código da aplicação. Para uma execução ainda mais eficiente, definimos as seguintes configurações:

- `Headless`: usado para desativar a geração da interface gráfica do navegador.

[1] https://www.docker.com/resources/what-container/

- `No-sandbox`: usado para garantir mais estabilidade no processo e evitar erros de privilégio.
- `disable-dev-shm-usage`: usado para forçar o navegador a usar a pasta `/temp` em vez da pasta padrão `/dev/shm`, usada em contêineres para arquivos compartilhados.
- `'images:2'`: usado para evitar que o navegador faça o download de imagens, uma vez que elas não serão necessárias para o experimento e podem causar consumo de recurso maior do que o necessário.

2.3.2 Configurações iniciais – Puppeteer

A configuração de ambiente para o *Puppeteer.js* foi, propositalmente, semelhante à configuração para *Selenium*: foi criado um contêiner usando como imagem base `node:14-slim`, que é a versão mais recomendada para uso no momento em que o experimento foi executado. Após isso, é feita a atualização de pacotes e instalação do *Google Chrome*, pois apesar de *Puppeteer.js* já vir com uma instância de navegador própria, o *Chromium*, instalar o *Google Chrome* garantirá mais estabilidade ao robô, pois irá garantir a presença das bibliotecas necessárias e pacotes de *Charset* para a maioria das línguas. Ao final, copiamos nosso repositório para o ambiente do contêiner e instalamos as dependências de código usando o gerenciador de pacotes *NPM*. As configurações de execução são as mesmas usadas no ambiente anterior, para garantir o máximo de semelhança entre os ambientes.

2.3.3 Automação de preenchimento de formulários

A automação de preenchimento de fomulários é largamente utilizada em grandes empresas, onde há grandes volumes de dados em trânsito e procedimentos padronizados. Através dela é possível alcançar grandes ganhos em produtividade, padronização e bem estar do funcionário [Willcocks, Lacity e Craig 2015]. O preenchimento de formulário é uma modalidade de automação aplicada em situações onde existem dados estruturados, interfaces padronizadas e é possível verificar o resultado determinando sucesso ou falha da inserção [Aguirre e Rodriguez 2017]. Preenchimento de formulários são usados quando há uma fonte de informações e um destino que receberá estas informações, mas não há uma integração direta entre eles. Nestes casos, uma pessoa qualificada deve ler as informações e fazer a interface, manualmente preenchendo os dados. Na maioria das vezes, tanto os dados quanto o formulário são padronizados, o que facilita a automação do processo. O algoritmo 1 trás uma rotina para essa situação.

2.3.4 Automação de captura de dados e geração de relatórios

Uma outra aplicação da automação, talvez a mais conhecida, se dá na captura e compilação de dados da web, prática conhecida com *web scrapping* ou *crawling*. As aplicações para este tipo de automação são inúmeras: desde a criação de bases de dados para estudos, geração de relatórios em empresas e até mesmo geração de índices de busca na web [Algiryage, Dias e Jayasena 2018]. Em nosso experimento, criamos um *Scrapper* com o objetivo de acessar as

Algoritmo 1: Preenchimento de formulários

Entradas: Lista de informações a serem inseridas em um formulário.
Saída: formulários preenchidos e submetidos.

```
1 Carregue a lista de informações;
2 Acesse a página da web;
3 Aguarde o carregamento;
4 para Cada entrada da lista faça
5     Preencha os dados no formulário;
6     Submeta o formulário;
```

páginas do evento de conferência *WebMedia*[2] e compilar dados das sessões técnicas para fins de estudo. O algoritmo 2 utilizado busca gerar um relatório das sessões técnicas apresentadas no evento.

Algoritmo 2: extração de dados de uma página web

Saída: Dados compilados e estruturados.

```
1  Acesse a página da web;
2  Aguarde o carregamento;
3  Liste todas as sessões técnicas;
4  Liste todos os parágrafos da página;
5  para cada parágrafo listado faça
6      Identificar chair da sessão;
7      Extrair título dos artigos;
8      Extrair autores dos artigos;
9  Compilar os dados em formato JSON;
10 Compilar os dados em formato CSV;
11 Escrever os dados em uma tabela;
```

Para fins de acurácia do experimento e redução de variáveis que possam causar incertezas na leitura do consumo de recursos foi suprimida a fase de escrita dos dados em disco, bem como o número de páginas visitadas se limitou a apenas uma, evitando assim incertezas causadas por demora de resposta por parte do servidor de cada página (que podem estar hospedadas em servidores diferentes) ou atrasos de pacotes. A página da edição WebMedia 2019[3] foi selecionada para este experimento por possuir o maior número de artigos e sessões técnicas, evidenciando de forma mais clara a existência de alguma diferença notável de desempenho.

[2] Simpósio Brasileiro de Sistemas Multimídia e Web
[3] Disponível em `https://webmedia.org.br/2019/programacao-em-html`

CAPÍTULO 3

Desenvolvimento

Neste capítulo faremos a comparação entre a biblioteca de node.JS, *Puppeteer.Js*, e a biblioteca de python, *Selenium*. Primeiramente, veremos aspectos de desenvolvimento, e em seguida faremos a comparação usando aspectos técnicos. Apesar de Selenium também ser compatível com node.JS, neste trabalho foi levado em conta a sua versão mais usada em processos de automação, Selenium Python. O fato de as bibliotecas virem de origens diferentes implica que seu desempenho estará relacionado também à linguagem de origem de suas bibliotecas.

3.1 Aspectos de desenvolvimento

Selenium e *Puppeteer* são nomes bem conhecidos no mundo da automação, ambas estão na linha de frente de qualquer desenvolvedor que se depara com a necessidade de criar uma solução para rotinas repetitivas na web, porém as duas não poderiam ter origens e histórias mais diferentes.

Selenium foi originalmente desenvolvido em 2004 como uma solução para testes automatizados em aplicações web. Originalmente, o nome dado à ferramenta foi *"JavaScriptTestRunner"*. A ferramenta logo ganhou fama na empresa que a desenvolveu, a ThoughtWorks, onde rapidamente surgiu a ideia de torná-la *open-source* e, posteriormente, de criar um *driver* que tornasse possível a conexão de qualquer linguagem de programação para o desenvolvimento de testes. Atualmente *Selenium* possui bibliotecas que permitem automação a partir de várias linguagens de programação, como Java, Python, Ruby, Node.JS e C#.

Por outro lado, *Puppeteer* foi lançado oficialmente em 2017 pela Google e rapidamente ganhou notoriedade entre os desenvolvedores, impulsionada pela ascensão do *Node.JS*. A principal proposta era de ser uma ferramenta funcional, prática, uma vez que dispensa a necessidade de um *webdriver*, e performática, pois a interação é feita diretamente como o navegador através do protocolo *DevTools*. A tabela 3.1 faz um comparativo de aspectos relevantes entre as duas ferramentas.

Por estar há mais tempo no mercado e sempre em constante atualização, *Selenium* possui uma vasta comunidade de usuários, e assim encontra-se com facilidade material de desenvolvimento e ajuda de outros programadores em fóruns e artigos. Em adição, o fato de ser *open-source* faz com que *bugs* reportados sejam rapidamente solucionados. Outro fator que impulsiona novas pessoas a aderirem ao uso do *Selenium* é o fato de a mesma possuir uma IDE (*Integrated Development Environment*) própria, chamada de *Selenium IDE*, onde é possível fazer o desenvolvimento e acompanhar a execução de scripts e interação com websites. Conforme mostrado na figura 3.1, é possível verificar a superioridade de buscas por *Selenium*

Feature	Pupeeteer	Selenium
Linguagem de programação	Node.JS	Múltiplas linguagens
Interface com o navegador	via protocolo *DevTools*	Via *webdriver*
Navegadores suportados	Qualquer que use o motor Blink, Mozila	Qualquer que use o motor Blink, Mozila, Safari, Edge e outros
Suporte Mobile	Não	Sim

Tabela 3.1: Aspectos relevantes para Puppeteer e Selenium

frente a *Puppeteer* em pesquisas do Google. O gráfico traz informações desde o ano de 2004, quando houve o lançamento do *Selenium*, no entanto, as buscas por *Puppeteer* realizadas antes de 2017 devem ser considerados residuais neste estudo por serem anterior ao ano lançamento da ferramenta, e pelo fato da palavra possuir um significado na língua inglesa.

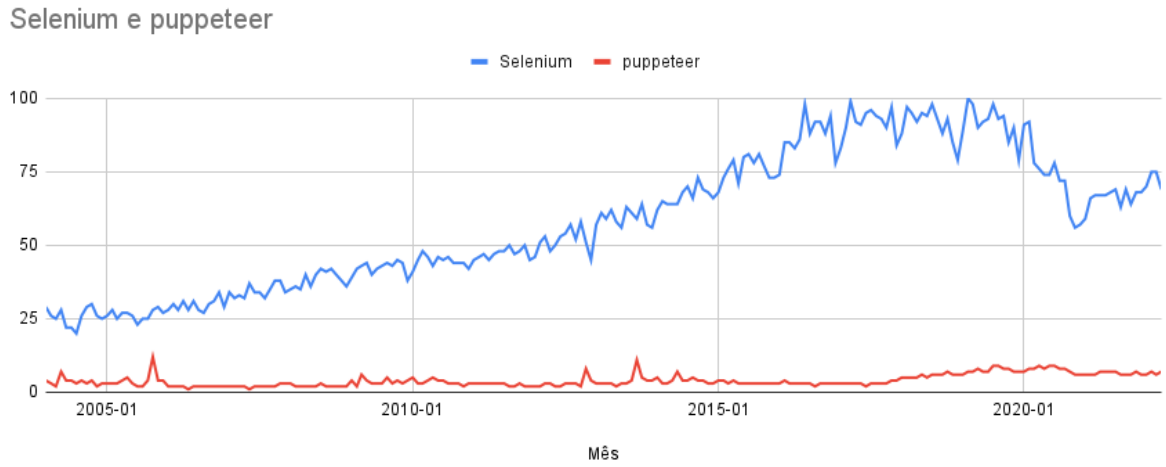

Figura 3.1: Pesquisas semanais evidenciam que Selenium é mais buscado na web que o puppeteer. Fonte: Google Trends

Para fazer frente ao Selenium, Puppeteer aposta em funcionalidades e praticidade. A biblioteca de Node.JS possui um vasto rol de métodos úteis para manipular navegador, *WebElements*, páginas, além de ser capaz de interceptar requisições do navegador, extrair e injetar códigos *JavaScript* dentro do contexto da página, definir *tracers*, lidar com extensões, fazer SSR (*Server-Side Rendering*) e muitas outras funcionalidades. Por usar diretamente o protocolo *Devtools* em vez de um *driver*, *Puppeteer* consegue extrair mais do navegador em um tempo menor sem comprometer tanto o desempenho.

3.2 Automação de preenchimento de formulários

O preenchimento de formulários de forma automática é por vezes o maior desejo de empresas e trabalhadores de escritórios. O trabalho repetitivo e pouco criativo leva à fadiga e desmotivação [Pan, Shell e Schleifer 1994]. Avaliamos nesta sessão os passos 5 e 6 do algorítimo 1, correspondente, respectivamente, ao preenchimento e submissão de dados em um formulário. Esses passos foram modularizados de modo que preenchimento e submissão estão em uma mesma função, sendo o trecho de código 3.1 correspondente ao preenchimento em *Puppeteer.Js* e o trecho 3.2 correspondente ao preenchimento em *Selenium*.

É possível observar que *Puppeteer.JS* se aproveita do assincronismo de javascript para sua construção. Boa parte das funções responde com *Promisses*, que apenas são resolvidas após a execução das mesmas. Várias destas *Promisses* podem estar em execução ao mesmo tempo, fazendo com que seja possível executar comandos de forma paralela com *Puppeteer*, o que porém não é o caso do código 3.1, onde cada comando é precedido da diretiva `'await'`, que explicitamente só permite continuar após a conclusão do comando.

```
/**
 * @description Fill data in a form
 * @param {puppeteer.Page} page
 * @param {Object} data
 */
const fillData = async (page, data) => {
  const input = sel => `input[ng-reflect-name="${sel}"]`
  await page.waitForSelector('input[value="Submit"]')
  await page.type(input("labelPhone"), data['Phone Number'])
  await page.type(input("labelLastName"), data['Last Name'])
  await page.type(input("labelEmail"), data['Email'])
  await page.type(input("labelCompanyName"), data['company'])
  await page.type(input("labelRole"), data['role'])
  await page.type(input("labelAddress"), data['Address'])
  await page.type(input("labelFirstName"), data['First Name'])
  await page.click('input[value="Submit"]')
}
```

Listing 3.1: Preenchendo formulário com Puppeteer.Js

Em ambos os códigos é possível verificar que a estratégia adotada foi a de buscar elementos da página usando seletores CSS, e, para cada elemento selecionado, enviar o dado corresponde recebido no argumento `data`, que define a informação a ser submetida no fomulário. É notável ainda que o código 3.1 é mais limpo que o 3.2, com métodos nomeados de forma mais simples e intuitiva. Isto acontece porque em *Selenium* é necessário sempre fazer uma busca pelo elemento o qual se deseja manipular usando `find_element` e `By.CSS_SELECTOR` e, então, realizar a manipulação com `send_keys`, ao passo que com *Puppeteer* é possível realizar estes passos apenas com o comando `type`.

```
SEL = By.CSS_SELECTOR

def fillData(driver: webdriver.Chrome, data):
  # Fill data in a form
  input = lambda sel: 'input[ng-reflect-name="{}"]'.format(sel)
```

```
  driver.find_element(SEL, 'input[value="Submit"]')
  driver.find_element(SEL,
        input('labelPhone')).send_keys(data["Phone Number"])
  driver.find_element(SEL,
        input('labelLastName')).send_keys(data["Last Name"])
  driver.find_element(SEL,
        input('labelEmail')).send_keys(data["Email"])
  driver.find_element(SEL,
        input('labelCompanyName')).send_keys(data["company"])
  driver.find_element(SEL,
        input('labelRole')).send_keys(data["role"])
  driver.find_element(SEL,
        input('labelAddress')).send_keys(data["Address"])
  driver.find_element(SEL,
        input('labelFirstName')).send_keys(data["First Name"])
  driver.find_element(SEL, 'input[value="Submit"]').click()
```

Listing 3.2: Preenchendo formulário com Selenium

3.3 Automação de extração de dados e geração de relatórios

Extração de dados é uma das principais aplicações de automação. Os termos *crawling* e *scrapping* se tornaram ainda mais populares após *Big Data*. Avaliamos nesta sessão os passos 3 a 9 do algoritmo 2, correspondentes à leitura da página, extração e compilação dos dados.

Em ambos os códigos 3.3 e 3.4, iniciamos o processo listando as sessões técnicas da página, para isso usamos `XPATH` para buscar os parágrafos contendo `'ST'` em seu conteúdo. Em seguida buscamos os títulos de cada sessão e prosseguimos listando todos os parágrafos da página. O objetivo é, utilizando a estrutura de organização das informações da página e tendo o número total de sessões técnicas, extrair corretamente títulos e autores dos artigos apresentados em cada uma das sessões e montar um *array* com estas informações.

```
/**
 * @description Extract the data from webmedia's website
 * @param {puppeteer.Page} page
 * @param {number} day
 * @param {string} strDate
 */
const getData = async (page, data) => {
  await page.waitForSelector('h4')
  const sessions = (await page.$x(`//p[contains(., 'ST')]`))[0]
  const sessionsNum = await sessions.$$eval('a', e => e.length)
  const sessionsNameTemp = await page.$$eval('h4', e => e.map((e) => e.
    textContent))
  const sessionsName = sessionsNameTemp.map(e => {
    if(e.includes('ST')) return e
  })
  const sessionsParagraph = await page.$$('p')
  let chair
  let actualSession
  let counter = 0
```

```
  for(const p of sessionsParagraph){
    let pText = (await p.evaluate(el => el.textContent)).replace(/\n/g, '')
   .replace('  ', ' ')
    if(pText?.includes('Chair') || pText?.includes('chair') ){
      actualSession = sessionsName[counter]
      chair = pText.replace('Chair: ', '').replace('chair: ', '')
      counter++
      if(counter > sessionsNum) {
        break
      }
    }else if(counter > 0){
      const title = (await p.$$eval('strong', e => e.map((e) => e.
   textContent))).join('').replace('\n', '')
      const author = pText.replace(title, '')
      sessionsArray.push({
        name: actualSession,
        chair,
        title,
        author
      })
    }
  }
  // printToFile(sessionsArray, data.year)
}
```

Listing 3.3: Extraindo sessões técnicas com Puppeteer.Js

Dentro do *loop* iteramos pelos parágrafos coletados, buscando por palavras-chave como `Chair` ou por padrões de construção, como a tag `strong`. A partir desses padrões usamos tratamentos de texto para separar a informação útil.

```
def getData(driver: webdriver.Chrome, year):
    sessions = driver.find_element(By.XPATH, "//p[contains(., 'ST')]")
    sessionsNum = len(sessions.find_elements(By.TAG_NAME, 'a'))
    sessionsNameTemp = driver.find_elements(By.TAG_NAME, 'h4')
    paragraphs = driver.find_elements(By.TAG_NAME, 'p')
    sessionsName = []
    for webElement in sessionsNameTemp:
        if "ST" in webElement.text:
            sessionsName.append(webElement.text)
    sessionsArray = []
    counter = 0
    for p in paragraphs:
        pText = p.text.replace('\n', '')
        if 'chair' in pText or 'Chair' in pText:
            if counter >= len(sessionsName):
                break
            actualSession = sessionsName[counter]
            chair = pText.replace('Chair: ', '').replace('chair: ', '')
            counter += 1
        elif counter > 0:
            title = ''
            for e in p.find_elements(By.TAG_NAME, 'strong'):
```

```
                title += e.text.replace('\n', '')
            author = pText.replace(title, '')
            sessionsArray.append({
                'name': actualSession,
                'chair': chair,
                'title': title,
                'author': author
            })
```

Listing 3.4: Extraindo sessões técnicas com Selenium

CAPÍTULO 4

Resultados

Neste capítulo serão apresentados os resultados obtidos a partir de execuções dos métodos apresentados anteriormente. A primeira parte trará resultados comparativos para a automação de preenchimento de formulários, e em seguida serão apresentados os resultados para a extração de dados e geração de relatórios.

O consumo medido se dá em valores percentuais para processamento (CPU), denotando o quanto está sendo requisitado de um core da CPU. Logo, um consumo de 100% de CPU significa que está sendo consumida em sua totalidade a capacidade de processamento de 1 core, 200% estaria, portanto consumindo 2 cores em sua totalidade, e assim por diante. O consumo de memória por sua vez é medido em mebibytes (Mib) e denota a quantidade de espaço alocado em tempo de execução.

4.1 Caso 1: Automação de preenchimento de formulários

A automação de preenchimento de formulário é uma atividade que envolve buscar pelos campos de um formulário e inserir a informação correspondente a ele, logo, trata-se de ler de uma variável ou arquivo e preencher na aplicação automatizada simulando um trabalho executado de forma manual. O gráfico 4.1 mostra o consumo médio de processamento medido em valores percentuais. Para a rotina automatizada em *Puppeteer.Js* tivemos um consumo médio de (162 $\pm$ 4)% de CPU, enquanto que o valor para a mesma rotina automatizada em *Selenium* foi de (158 $\pm$ 9)%, o que implica igualdade estatística.

O consumo de memória pode ser visto no gráfico 4.2. O valor observado foi de (143 $\pm$ 7) MiB para a rotina desenvolvida em *PuppeteerJs* e (145 $\pm$ 4) MiB para a rotina desenvolvida em *Selenium*. Mais uma vez, os valores médios das execuções estão dentro do intervalo de incerteza, demonstrando igualdade estatística.

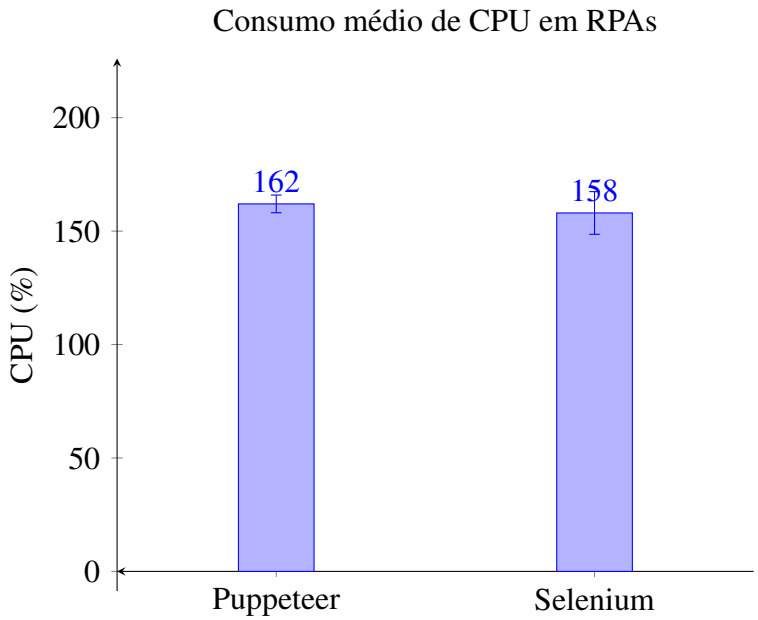

Figura 4.1: Consumo médio de CPU(%) em RPAs

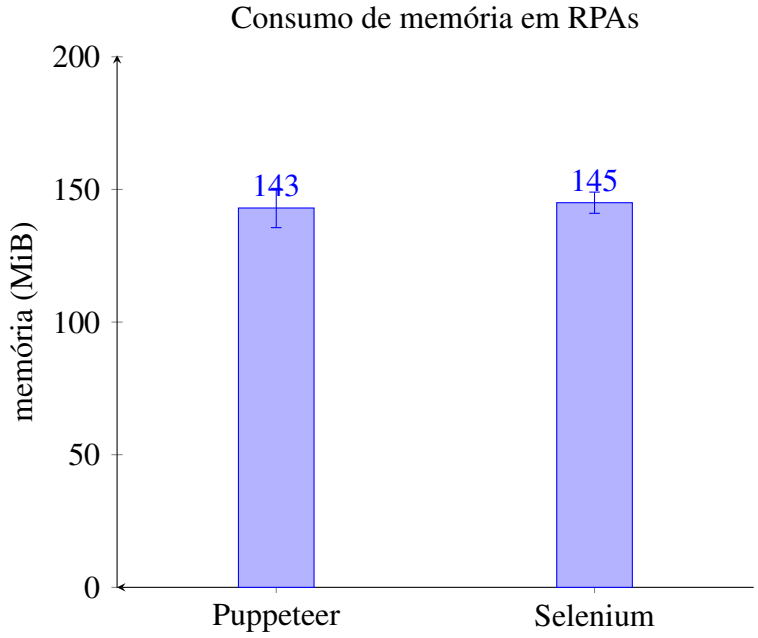

Figura 4.2: Consumo de memória(MiB) em RPAs

4.2 Caso 2: Automação de extração de dados e geração de relatórios

Extração de dados é uma atividade que envolve acessar páginas da web com o objetivo de capturar e estruturar informações. É uma prática muito utilizada para a construção de bases de dados, geração de relatórios e até mesmo para indexação de sites na web.

Diferentemente do experimento anterior, onde foi observada igualdade estatística nas execuções, o gráfico 4.3 evidencia uma grande disparidade entre o consumo médio de CPU para a extração de dados usando *Puppeteer.JS* (6 $\pm$ 1)% e a mesma situação usando *Selenium* (109 $\pm$ 1)%, evidenciando uma superioridade de desempenho ao se utilizar *Puppeteer.JS*.

A discrepância de valores para deste experimento, ilustrada no gráfico da figura 4.3, pode ser explicada pela diferença de construção das ferramentas analisadas. *Puppeteer.Js* foi construída a partir de tecnologias mais recentes e com maior compatibilidade com o navegador utilizado, o *Chrome*. Outro fator que pode ser determinante na diferença de desempenho é a forma como é feita a interface entre o código escrito e o navegador; o protocolo *DevTools*, nativo dos navegadores baseados em *Chrome*, torna a interface entre navegador e código muito mais leve em comparação com a utilização de um *webdriver*.

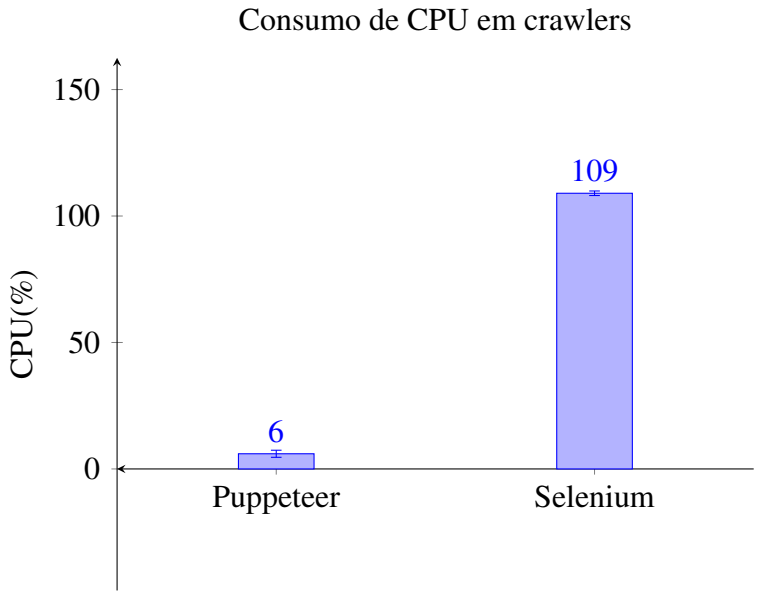

Figura 4.3: Consumo de CPU(%) em crawlers

De forma semelhante, o gráfico 4.4 nos mostra uma diferença considerável no consumo de memória entre ambas as ferramentas, com o *Puppeteer* apresentando um consumo médio de memória de (99 $\pm$ 2) MiB, enquanto *Selenium* obteve um consumo de (138 $\pm$ 4) MiB, reforçando a superioridade de desempenho por parte do *Puppeteer.JS* para a construção de *WebCrawlers*.

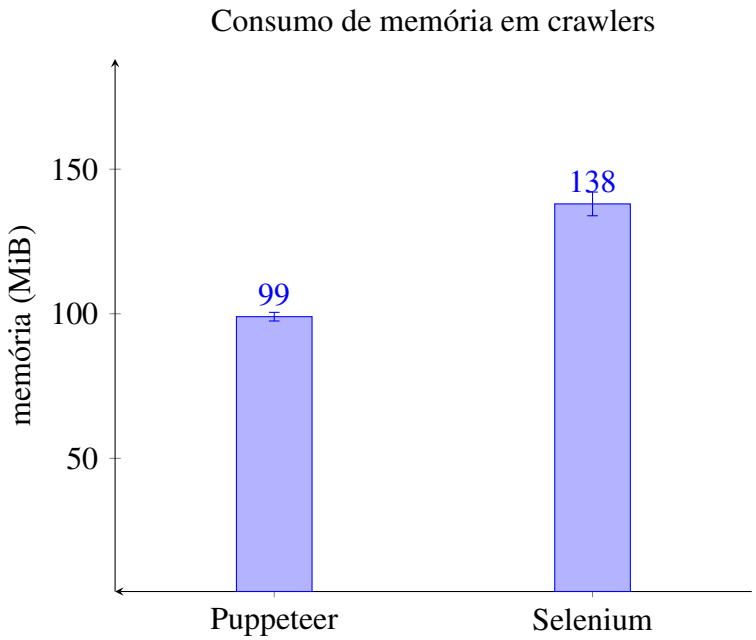

Figura 4.4: Consumo de memória em crawlers

CAPÍTULO 5

Conclusões e Trabalhos Futuros

Neste trabalho, foi implementada uma série de experimentos de automação de processos com o objetivo de comparar o desempenho de duas bibliotecas utilizadas para a construção de robôs: *PuppeteerJs* e *Selenium*. Os casos de uso envolveram automatizar o preenchimento e submissão de formulários, e também a extração e estruturação de dados, dois dos mais comuns casos de uso, onde se aplica a automação devido à repetitividade do processo. As implementações foram avaliadas de maneira quantitativa, identificando diferenças no consumo de recursos.

Os resultados obtidos mostraram que para o caso de automação de preenchimento de formulários, ambas as ferramentas apresentam desempenho semelhante para consumo de memória e capacidade de processamento. O mesmo resultado, porém, não é obtido quando aplicamos as ferramentas em um trabalho de *WebScraping*, onde podemos observar uma grande diferença de desempenho favorecendo a ferramenta *PuppeteerJs*.

A diferença entre os valores observados nos dois experimentos, por sua vez, é fruto da natureza da ação realizada: o preenchimento de formulários demanda muito mais recursos da aplicação, uma vez que é preciso simular periféricos como *clicks* do mouse e entradas de teclado, e, mesmo envolvendo menos seletores, foram observados valores maiores de consumo de recursos. Já a extração de dados envolve percorrer todos os elementos de uma página e fazer leitura de seus valores, uma atividade bem menos onerosa para a máquina. A diferença gritante por sua vez entre os valores observados para as duas execuções deve-se à quantidade de seletores lidos na página. Os experimentos realizados no trabalho em questão foram úteis para elucidar diferenças de desempenho para as ferramentas de automação. Entretanto, outros casos de uso e variáveis precisam ser observados para uma conclusão mais assertiva.

Por fim, trabalhos futuros devem incluir:

- A medição do tempo de execução nos experimentos;
- A utilização de outros recursos disponíveis nas ferramentas, como a impressão e a captura de tela; e
- Utilizar formulários maiores com mais riqueza de entrada de dados.

Referências Bibliográficas

AGUIRRE, S.; RODRIGUEZ, A. Automation of a business process using robotic process automation (rpa): A case study. In: FIGUEROA-GARCÍA, J. C. et al. (Ed.). *Applied Computer Sciences in Engineering*. Cham: Springer International Publishing, 2017. p. 65–71. ISBN 978-3-319-66963-2.

ALGIRYAGE, N.; DIAS, G.; JAYASENA, S. Distinguishing real web crawlers from fakes: Googlebot example. In: *2018 Moratuwa Engineering Research Conference (MERCon)*. [S.l.: s.n.], 2018. p. 13–18.

ANAGNOSTE, S. Robotic automation process - the next major revolution in terms of back office operations improvement. *Proceedings of the International Conference on Business Excellence*, v. 11, 07 2017.

GARCíA, B. et al. A survey of the selenium ecosystem. *Electronics*, v. 9, n. 7, 2020. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/9/7/1067>.

HAT, I. R. *Containers Linux*. 2019. Disponível em: `https://www.redhat.com/pt-br/topics/containers`.

HINDEL, J.; CABRERA, L. M.; STIERLE, M. Robotic process automation: Hype or hope? *Band-1*, 2020.

INC, G. *Chrome DevTools Protocol*. 2020. Disponível em: `https://chromedevtools.github.io/devtools-protocol/`.

KAYA, C.; TURKYILMAZ, M.; BIROL, B. Impact of rpa technologies on accounting systems. *Muhasebe ve Finansman Dergisi*, p. 235–250, 04 2019.

LLC, G. *Googlebot*. 2022. Disponível em `https://developers.google.com/search/docs/advanced/crawling/googlebot`.

LLC, G. *How Google Search Works*. 2022. Disponível em `https://developers.google.com/search/docs/beginner/how-search-works`.

LLC, G. *Puppeteer*. 2022. Disponível em `https://developers.google.com/web/tools/puppeteer`.

MEIRONKE, A.; KUEHNEL, S. How to measure rpa's benefits? a review on metrics, indicators, and evaluation methods of rpa benefit assessment. In: *Wirtschaftsinformatik 2022 Proceedings. 5*. Wirtschaftsinformatik, 2022. Disponível em: <https://aisel.aisnet.org/wi2022/bpm/bpm/5>.

MITCHELL, R. *Web Scraping with Python*. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc., 2018.

PAN, C. S.; SHELL, R. L.; SCHLEIFER, L. M. Performance variability as an indicator of fatigue and boredom effects in a vdt data-entry task. *International Journal of Human–Computer Interaction*, Taylor and Francis, v. 6, n. 1, p. 37–45, 1994. Disponível em: <https://doi.org/10.1080/10447319409526082>.

PERSSON, E. *Evaluating tools and techniques for web scraping*. 2019.

SELENIUM. 2022. Disponível em `https://www.browserstack.com/selenium`.

SIDERSKA, J. The adoption of robotic process automation technology to ensure business processes during the covid-19 pandemic. *Sustainability*, v. 13, n. 14, 2021. ISSN 2071-1050. Disponível em: <https://www.mdpi.com/2071-1050/13/14/8020>.

WILLCOCKS, L.; LACITY, M.; CRAIG, A. Robotic process automation at telefónica o2. *MIS Q Exec*, v. 15, n. 1, p. 21–35, 2015.

WILLCOCKS, L.; LACITY, M.; CRAIG, A. Robotic process automation at xchanging. *MIS Q Exec*, v. 15, 2015.