Universidade Federal de Roraima
Centro de Ciências e Tecnologias
Departamento de Ciência da Computação
Curso Ciências da Computação
Sistemas Operacionais – DCC403

Relatório da Construção de um Navegador Web Minimalista com Foco em Conceitos de Sistemas Operacionais

Boa vista, Roraima. 06 de agosto de 2025

Universidade Federal de Roraima
Centro de Ciências e Tecnologias
Departamento de Ciência da Computação
Curso Ciências da Computação
Sistemas Operacionais – DCC403

Discentes: Lucas Costancio e Yan Teixeira

Relatório da Construção de um Navegador Web Minimalista com Foco em Conceitos de Sistemas Operacionais

Esse é o relatório do projeto final da disciplina de Sistema Operacionais, do curso Ciências da computação, da UFRR (Universidade Federal de Roraima), como requisito parcial para obtenção de nota na disciplina.

Orientador(a): Prof. Dr. Hebert Oliveira Rocha

Boa vista, Roraima. 06 de agosto de 2025

Resumo

O projeto consistiu no desenvolvimento de um navegador web leve utilizando Electron e as linguagens JavaScript, HTML e CSS, com foco em gerenciamento de recursos e segurança. Foram implementados processos isolados para cada aba, monitoramento e controle de uso de CPU e memória, recursos de navegação (botões, barra de endereços, bookmarks, histórico, múltiplas abas) e técnicas de sandboxing para isolamento de processos. A execução e visualização do projeto foram realizadas no terminal do Linux. O sistema foi validado por meio de testes de desempenho, estabilidade, rede, compatibilidade e segurança.

Palavra-chave: Navegador web, Electron, Gerenciamento de recursos CPU e memória, Sandboxing, Segurança, Multiprocessamento

Sumário

1.Introdução

1.1 Justificativa

A crescente demanda por navegadores capazes de equilibrar desempenho, segurança e baixo consumo de recursos torna essencial o estudo e desenvolvimento de soluções que integrem esses aspectos. Este projeto, ao unir tecnologias modernas como Electron, JavaScript, HTML e CSS, com práticas de gerenciamento de processos e isolamento de recursos, proporciona um ambiente de testes e aplicação prática dos conceitos estudados na disciplina de Sistemas Operacionais. Além disso, a implementação em ambiente Linux e o uso do terminal para visualização reforçam o aprendizado sobre a interação entre software e sistema.

1.2 Objetivo Geral

Desenvolver um **navegador web leve**, capaz de oferecer recursos básicos de navegação e controle eficiente de recursos do sistema, garantindo segurança e estabilidade.

1.3 Objetivos Específicos

- Implementar recursos de navegação como barra de endereços, botões de controle, histórico, bookmarks e múltiplas abas.
- Utilizar técnicas de multiprocessamento, criando processos isolados para cada aba.
- Monitorar em tempo real o uso de CPU e memória, com mecanismos de controle quando os limites forem excedidos.
- Aplicar técnicas de sandboxing para segurança e isolamento de processos.
- Realizar testes de desempenho, estabilidade, rede, compatibilidade e segurança.

2 Fundamentação Teórica

O desenvolvimento de navegadores web envolve a integração de múltiplos componentes responsáveis pela interpretação, renderização e exibição de páginas da internet, além da comunicação com servidores via protocolos de rede. A escolha das bibliotecas e frameworks utilizados influencia diretamente no desempenho, segurança e recursos oferecidos pela aplicação.

2.1 Conceitos de navegadores web

Um navegador web é um software projetado para acessar, interpretar e exibir conteúdos provenientes da World Wide Web. Sua principal função é processar linguagens como HTML, CSS e JavaScript, convertendo-as em interfaces visuais interativas. Os motores de renderização, como o **Blink** (utilizado pelo Chromium e pelo Electron), desempenham papel central nessa tarefa, sendo responsáveis por transformar o código recebido em elementos gráficos visíveis ao usuário.

2.2 Gerenciamento de processos e recursos

Em sistemas operacionais modernos, o gerenciamento de processos e recursos é fundamental para garantir estabilidade e eficiência. Aplicações que abrem múltiplas abas, como navegadores, podem adotar um modelo **multiprocessado**, no qual cada aba é executada em um processo independente. Essa abordagem melhora a segurança e evita que falhas em uma aba comprometam o funcionamento das demais. O monitoramento de **uso de CPU e memória** permite detectar situações de sobrecarga, possibilitando a adoção de medidas preventivas, como congelamento temporário ou encerramento do processo problemático.

2.3 Sandboxing e segurança

A segurança em navegadores é garantida por meio de técnicas como o **sandboxing**, que consiste em isolar processos para impedir que código malicioso acesse recursos críticos do sistema operacional. Esse isolamento reduz riscos de exploração de vulnerabilidades, limita o acesso a arquivos e impede que scripts não autorizados interfiram em outras abas ou no funcionamento geral do navegador.

2.4 Bibliotecas e tecnologias utilizadas

O projeto faz uso do **Electron**, framework que combina o motor de renderização do Chromium com o ambiente Node.js, permitindo o desenvolvimento de aplicações desktop utilizando **JavaScript, HTML e CSS**. Essa combinação possibilita criar interfaces gráficas modernas e interativas, ao mesmo tempo em que se integra a funcionalidades nativas do sistema operacional. A execução e testes foram realizados em **Linux**, aproveitando o terminal para visualização de logs e monitoramento em tempo real.

3 Metodologia

O desenvolvimento do projeto foi realizado com base na integração de tecnologias modernas para criação de aplicações desktop, combinando conceitos de Sistemas Operacionais com recursos de programação web. A metodologia adotada envolveu desde a definição do ambiente de desenvolvimento até a implementação modular do sistema, garantindo organização e facilidade de manutenção.

3.1 Ferramentas e linguagens utilizadas

As Ferramentas e linguagens de programação utilizadas nesse projeto foram:

- **Electron (versão 8.0.0)** – Framework que combina o motor de renderização do Chromium com Node.js, permitindo o desenvolvimento de aplicações desktop utilizando tecnologias web.

- **Node.js** – Ambiente de execução JavaScript no lado do servidor, utilizado para comunicação com APIs e interação com recursos do sistema operacional.
  JavaScript, HTML e CSS – Linguagens utilizadas para a lógica de funcionamento, estrutura e estilização da interface do navegador.

- **Linux** – Sistema operacional utilizado para desenvolvimento e testes, com apoio do terminal para execução, monitoramento e depuração do projeto.

3.2 Ambiente de desenvolvimento

O projeto foi estruturado de forma modular, com arquivos e funções organizados para separar responsabilidades:

- **main.js** – Arquivo principal responsável por inicializar o aplicativo, criar janelas e gerenciar processos.
- **package.json** e **package-lock.json** – Arquivos de configuração do projeto e gerenciamento de dependências.
- **preload.js** – Script intermediário entre o processo principal e o renderizador, garantindo segurança no acesso a recursos do sistema.
  seguranca.js – Módulo responsável pela aplicação de políticas de segurança e isolamento (sandboxing).
- **historico.html** – Página para exibição do histórico de navegação.
- **index.html** – Página principal do navegador, contendo a interface e os elementos de navegação.

3.3 Estrutura do sistema e módulos

O processo de desenvolvimento seguiu as seguintes etapas:

1. **Configuração do ambiente** com instalação do Node.js e Electron na versão 8.0.0.
2. **Estruturação do projeto** em módulos independentes para facilitar manutenção e testes.
3. **Implementação da interface gráfica** utilizando HTML, CSS e integração com JavaScript.
4. **Desenvolvimento das funcionalidades de navegação** (barra de endereços, botões, múltiplas abas, histórico, bookmarks).
5. **Implementação de monitoramento de recursos** (CPU e memória) e mecanismos de bloqueio quando limites são ultrapassados.
6. **Aplicação de técnicas de sandboxing** para isolamento de processos e aumento da segurança.

7. **Testes de desempenho, estabilidade, rede, compatibilidade e segurança** no ambiente Linux.

4.Desenvolvimento

O desenvolvimento do navegador foi realizado de forma modular, permitindo que cada funcionalidade fosse implementada e testada separadamente. A integração final resultou em um sistema funcional, leve e seguro, seguindo as especificações propostas.

4.1 Implementação dos recursos de navegação

A interface principal foi construída no arquivo **index.html**, utilizando HTML para a estrutura, CSS para o design visual e JavaScript para a lógica de funcionamento. Foram implementados:

Barra de endereços: foi inspirada no design do Google Chrome e Firefox, sendo posicionada tanto na parte superior **(`top-bar`)** quanto no centro da tela inicial **(`home-screen`).** A navegação é controlada pela **function `go()`,** que captura o texto digitado na barra e o envia para o processo principal por meio da **`electronAPI`**. Essa API, definida no **`preload.js`,** utiliza o **`ipcRenderer`** para comunicar com o **`main.js`**, que carrega a URL correspondente na aba ativa.

FIGURA I

```
function go() {
    const url = document.getElementById("url").value;
    window.electronAPI.navigate(url);
}
```

O **`preload.js`** expõe essa função de forma segura:

FIGURA II

```
contextBridge.exposeInMainWorld('electronAPI', {
  navigate: (url) => ipcRenderer.send('navigate', url),
```

Botões de navegação: os três botões foram adicionados no **`index.html`** para controlar a navegação da aba atual: voltar, avançar e recarregar. Cada botão dispara um evento para o processo principal via API do preload.

FIGURA III

```
</div>
<button onclick="window.electronAPI.back()">⬅</button>
<button onclick="window.electronAPI.forward()">➡</button>
<button onclick="window.electronAPI.reload()">🔄</button>
<button id="history-btn" onclick="window.electronAPI.showHistory()">Histórico</button>
```

No arquivo **preload.js** esses comandos estão estruturados dessa forma:

```
contextBridge.exposeInMainWorld('electronAPI', {
    navigate: (url) => ipcRenderer.send('navigate', url),
    back: () => ipcRenderer.send('back'),
    forward: () => ipcRenderer.send('forward'),
    reload: () => ipcRenderer.send('reload'),
```

Cada uma dessas funções envia uma **mensagem** para o processo principal (`main.js`) usando o **`ipcRenderer.send()`,** que é o mecanismo de **comunicação entre processos** do Electron:

- **`back`:** solicita que o navegador volte à página anterior na aba ativa.
- **`forward`:** solicita que avance para a próxima página, se houver.
- **`reload`:** solicita o recarregamento da página atual.

No arquivo **`main.js`**, foram implementados os **listeners de eventos IPC** para controlar as ações de navegação no navegador, como voltar, avançar e recarregar a página atual. Esses eventos são disparados pela interface gráfica (via **`preload.js`**) e tratados no processo principal utilizando o módulo **`ipcMain`** do Electron.

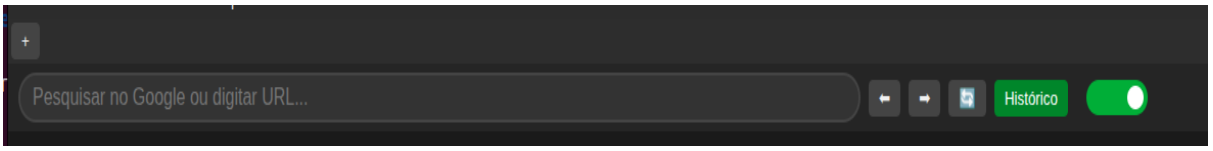
FIGURA V

```
ipcMain.on('back', () => {
    if (tabs[currentTab]?.webContents.navigationHistory.goBack()) {
        tabs[currentTab].webContents.goBack();
    }
});

ipcMain.on('forward', () => {
    if (tabs[currentTab]?.webContents.navigationHistory.goForward()) {
        tabs[currentTab].webContents.goForward();
    }
});

ipcMain.on('reload', () => {
    tabs[currentTab]?.webContents.reload();
});
```

- **ipcMain.on(‘back’):** Esta função verifica se a aba atual possui histórico anterior com `navigationHistory.goBack()`. Se houver, executa o método `.goBack()` do `webContents`, retornando à página anterior.
- **ipcMain.on('forward'):** Semelhante à função de voltar, verifica se há uma próxima página no histórico e, se disponível, utiliza **`.goForward()`** para navegar adiante.
- **ipcMain.on('reload')** Recarrega o conteúdo da aba ativa utilizando **`.reload()`**, sem alterar o histórico de navegação.

Com a estrutura do projeto devidamente definida, procede-se à organização do arquivo **index,** de modo a replicar com fidelidade a interface e o comportamento de um navegador convencional.

FIGURA VI



Sistema de histórico: é gerenciado no **`main.js`** e exibido no arquivo **`historico.html`.** Cada página visitada é registrada com título, URL e data/hora, e pode ser visualizada em uma lista interativa**,** graças a **async function loadHisory()** que permite a visualização do histórico do navegador.

FIGURA VII

```
async function loadHistory() {
    const history = await window.electronAPI.getHistory();
    const list = document.getElementById('history-list');

    if (history.length === 0) {
        list.innerHTML = '<div class="empty-history">Nenhum item no histórico.</div>';
        return;
    }

    list.innerHTML = history.map(item => `
        <div class="history-item" onclick="navigateTo('${item.url}')">
            <div class="history-info">
                <div class="history-title">${item.title}</div>
                <div class="history-url">${item.url}</div>
            </div>
            <div class="history-time">${new Date(item.timestamp).toLocaleString()}</div>
        </div>
    `).join('');
}
```

E possui a função de limpeza de histórico:

FIGURA VIII

```
async function clearHistory() {
    await window.electronAPI.clearHistory();
    loadHistory();
}
```

Essa função:

1. Envia uma solicitação ao processo principal via **`electronAPI.clearHistory()`**;
2. O `main.js` então limpa o array de histórico:
3. Por fim, a interface é recarregada para mostrar que não há mais itens salvos.

FIGURA-9

```
ipcMain.on('clear-history', () => {
    clearHistory();
});
```

Suporte a múltiplas abas: O navegador foi projetado com suporte a **múltiplas abas**, permitindo que o usuário abra, feche e alterne entre diversas páginas ao mesmo tempo, de forma similar aos navegadores modernos como Chrome e Firefox. Cada aba funciona de forma independente e é representada por um objeto do tipo **`BrowserView`** (com seu próprio **`webContents`**), isolando o conteúdo carregado.

O sistema de abas foi implementado por meio da colaboração entre os arquivos `main.js`, **`preload.js`** e **`index.html`**, utilizando o sistema de **IPC (Inter-Process Communication)** do Electron para troca de mensagens entre o processo principal e o processo de interface.

```
let tabs = [];
let currentTab = 0;
```

Cada elemento do array `tabs` contém um objeto com:

- **`view`:** instância de `BrowserView` que representa a aba visualmente.
- **`title`**: título da aba.
- **`url`:** URL atual da aba.

Criar Nova Aba

Ao clicar no botão "+" da interface, o navegador envia um evento `new-tab` por IPC:

`index.html`:

```
<div id="tab-bar">
    <button onclick="window.electronAPI.newTab()">+</button>
</div>
```

`preload.js`:

```
newTab: () => ipcRenderer.send('new-tab'),
```

Main.js:

```
ipcMain.on('new-tab', () => {
    createNewTab('home');
});

ipcMain.on('switch-tab', (event, index) => {
    switchTab(index);
});

ipcMain.on('close-tab', (event, index) => {
    closeTab(index);
});
```

`new-tab`: Cria uma nova aba utilizando **`BrowserView`**, carrega a URL inicial e adiciona à lista de abas.

`switch-tab`: Alterna a aba ativa exibida na janela principal.

`close-tab`: Fecha uma aba específica e reorganiza as abas restantes.

Atualizar Visualmente as Abas

A função **`updateTabs()`** no `main.js` envia à interface as informações das abas:

```
function updateTabs() {
    const tabList = tabs.map((_, i) => ({ index: i, active: i === currentTab }));
    mainWindow.webContents.send('update-tabs', tabList);
}
```

A função **`updateTabs()`** tem como objetivo manter a **interface da barra de abas sincronizada** com o estado atual das abas abertas no navegador. Ela é definida no processo principal **(`main.js`)** e é chamada sempre que uma aba é criada, fechada ou alterada.

Explicação dos componentes:

- **`tabs.map((_, i) => ({ index: i, active: i === currentTab })):`** percorre todas as abas e cria uma lista (`tabList`) com objetos que contêm:
    - **`index`:** o índice numérico de cada aba;
    - **`active`:** um valor booleano (`true` ou `false`) indicando se a aba está ativa (ou seja, sendo exibida no momento).
- **`mainWindow.webContents.send('update-tabs', tabList):`** envia o array `tabList` para o processo de interface (`index.html`) através do canal `'update-tabs'`.

Resultado na interface:

Ao receber esses dados, o front-end recria visualmente a barra de abas, destacando a aba ativa com base no valor de **`active`.** Assim, a interface se mantém coerente com o estado interno do navegador.

4.2 Multiprocessamento e isolamento por abas

O navegador foi projetado com suporte ao modelo **multiprocessado**, no qual cada aba é implementada como uma instância independente de `BrowserWindow`. Isso garante que cada aba opera-se em seu próprio processo de renderização, aumentando a segurança, estabilidade e desempenho da aplicação.

A arquitetura adotada permite que o conteúdo de cada aba funcione de forma isolada, sem interferir nas demais. A criação dessas abas é feita no processo principal, com configurações de segurança como **`contextIsolation:true e nodeIntegration: false`,** prevenindo que páginas da web acessem diretamente APIs do Node.js ou módulos do sistema.

Quando o usuário clica no botão de nova aba, é enviado um comando via IPC que aciona a criação da aba no processo principal. Esse processo já foi detalhado anteriormente, na seção referente ao suporte a múltiplas abas (ver Seção 4.1).

Troca de Abas e Função `switchTab()`

A alternância entre abas ocorre por meio da função **`switchTab()`**, responsável por ocultar todas as abas abertas e exibir apenas a aba atualmente ativa. Essa função percorre o array de abas abertas e usa os métodos `.hide()` e `.show()` para controlar a visibilidade das janelas.

```
function switchTab(index) {
    if (tabs[index]) {
        currentTab = index;
        mainWindow.setBrowserView(tabs[currentTab]);
        setTabBounds();
        updateTabs();
    }
}
```

O funcionamento é o seguinte:

Verificação de Existência:

Primeiro, a função verifica se existe uma aba no índice informado (tabs[index]). Isso evita erros caso o índice seja inválido.

Atualização do Índice da Aba Atual:

Se a aba existir, a variável **currentTab** é atualizada para o novo índice, indicando qual aba está ativa no momento.

Exibição da Aba Selecionada:

O método **mainWindow.setBrowserView(tabs[currentTab])** é chamado para exibir a aba selecionada na janela principal do navegador.

Ajuste do Tamanho da Aba:

A função s**etTabBounds()** é chamada para garantir que a nova aba ativa ocupe corretamente o espaço disponível na janela, respeitando o layout da interface.

Atualização Visual das Abas:

Por fim, a função **updateTabs()** é chamada para atualizar a interface gráfica, destacando a aba ativa e sincronizando o estado das abas com o frontend.

4.3 Monitoramento e controle de CPU/RAM

Com o objetivo de aplicar conceitos de gerenciamento de recursos em sistemas operacionais, o navegador desenvolvido implementa um sistema de **monitoramento contínuo do consumo de CPU e memória RAM**. Esse controle foi realizado por meio do arquivo `monitoramento.js`, que executa funções específicas para acompanhar o uso de recursos e aplicar restrições em tempo real.

Monitoramento de Recursos

No arquivo **`monitoramento.js`**, foi criada uma rotina periódica utilizando `setInterval`, que realiza a leitura do uso de CPU e memória do sistema a cada determinado intervalo de tempo. As informações são capturadas usando a biblioteca `os`, nativa do Node.js.

As porcentagens de consumo são exibidas diretamente no **terminal do Linux**, facilitando o diagnóstico e testes durante a execução do navegador.

```
> codigos-do-site@1.0.0 start
> electron .

[Monitoramento] CPU: 6.00% | RAM: 69.80%
```

Explicação da Função de Monitoramento

A função principal do arquivo é `monitoramento(app, win, dialog)`. Abaixo estão seus principais componentes:

- **1. Coleta de uso de CPU**: Utiliza o comando `top` do terminal para capturar a porcentagem da CPU em uso, através de `exec()`. O valor é processado com expressões regulares para extrair a parte relevante da saída do terminal.
- **2. Coleta de uso de Memória RAM**: Executa o comando `free -m` e calcula a porcentagem de memória utilizada com base nos valores de total e usado, extraídos da linha contendo a palavra "mem".

```
// cpuRaw é número (porcentagem)
const cpuMatch = stdout.match(/(\d+\.\d+)%?\s*id/);
const cpuRaw = cpuMatch ? (100 - parseFloat(cpuMatch[1])) : NaN;

exec("free -m", (err, memOut) => {
    if (err) {
        console.error('Erro ao coletar a memória:', err);
        return;
    }
```

Executa o comando **`top`** no terminal e extrai o valor de CPU ociosa **(`id`)**. Subtrai de 100 para obter a **porcentagem de CPU em uso real**.

FIGURA - Calculo da memória RAM

```
try {
    // Processa a saída de memória
    const lines = memOut.split('\n');
    const memLine = lines.find(line => line.toLowerCase().includes('mem'));
    if (!memLine) throw new Error('Linha de memória não encontrada');

    const memValues = memLine.trim().split(/\s+/);
    const total = parseInt(memValues[1], 10);
    const used = parseInt(memValues[2], 10);
    const memRaw = (used / total) * 100;
```

- **3. Exibição no terminal**: O consumo de CPU e RAM é exibido no terminal em tempo real com `process.stdout.write(...)`, substituindo a linha anterior para manter a tela limpa.

```
// Atualiza linha do console (visual)
process.stdout.clearLine(0);
process.stdout.cursorTo(0);
process.stdout.write(`\r[Monitoramento] CPU: ${cpuFormatted}% | RAM: ${memFormatted}% `);
```

Exibe o uso de CPU/RAM em **tempo real no terminal**, reescrevendo a mesma linha sem gerar múltiplas saídas.

Controle de Limites e Ações Preventivas

Além do monitoramento passivo, foi implementado um mecanismo de **controle de limites**: quando o consumo de CPU ou RAM ultrapassa um valor definido (ex: 80%), o sistema executa uma ação de segurança.

Essa ação consiste em enviar uma mensagem ao processo principal ou à interface, fazendo com que o navegador exiba uma **tela de alerta** com a opção de **fechar ou continuar** com o

navegador. Isso simula um sistema de proteção contra sobrecarga, comum em navegadores modernos.

Exibição do Alerta no Navegador

A comunicação entre o **`monitoramento.js`** e o navegador foi feita por meio de IPC (ou chamada direta dependendo da arquitetura). Quando o limite é ultrapassado, é disparado um evento que resulta na exibição de uma **janela modal com aviso de alto consumo**, permitindo ao usuário decidir se deseja continuar ou encerrar o navegador.

Esse alerta é importante tanto para **a usabilidade** quanto para a **estabilidade do sistema**, evitando que o navegador consuma recursos excessivos sem controle.

```
    // Alerta
    if (!alertando && (cpuRaw > 90 || memRaw > 80)) {
        alertando = true;
        if (win && win.webContents) win.webContents.send('congelar');

        const response = dialog.showMessageBoxSync(win, {
            type: 'warning',
            buttons: ['Fechar Navegador', 'Cancelar'],
            defaultId: 0,
            cancelId: 1,
            title: 'Alerta de Recursos',
            message: 'Uso de recursos elevado',
            detail: `Uso de CPU: ${cpuFormatted}% | RAM: ${memFormatted}%`
        });

        if (response === 0) {
            app.quit();
        } else {
            alertando = false;
        }
    }

} catch (parseErr) {
    console.error('Erro ao analisar saída de memória:', parseErr);
}
```

- Se o uso de CPU passar de **90%** ou RAM de **80%**, exibe **alerta modal** no navegador.
- O usuário pode **fechar o navegador** ou ignorar o alerta.

- Isso garante que o sistema continue estável, mesmo sob uso excessivo de recursos.

5.Testes e validação

A fase de testes teve como objetivo garantir que o navegador se comportasse de forma eficiente, estável e segura em diferentes situações. Os testes foram realizados manualmente e, quando possível, validados por saída no terminal e arquivos de log. A seguir, estão os principais testes aplicados:

5.1 Testes de desempenho

Para avaliar o desempenho do navegador, foram realizados testes práticos com base em quatro situações distintas: carregamento da página inicial, acesso ao Google, acesso ao YouTube e recarregamento da página inicial. Durante cada teste, foram coletadas as seguintes métricas:

- **Tempo de carregamento** da página (em milissegundos).

- **Consumo de CPU e RAM antes e depois do carregamento**, utilizando o sistema de monitoramento implementado no projeto.

Resultados dos testes:

Página Carregada	Tempo (ms)	CPU Inicial	CPU Pós-Carregamento	RAM Inicial	RAM Pós-Carregamento
Página Inicial	229	-	2.90%	-	71.67%
Google	1324	2.90%	9.10%	71.67%	72.41%
YouTube	3915	9.10%	16.70%	72.41%	75.74%
Página Inicial (reload)	1019	16.70%	4.50%	75.74%	75.74%

Análise dos dados:

- A **página inicial**, por ser leve, teve carregamento rápido (229 ms) e consumo mínimo de CPU.
- Ao acessar o **Google**, o tempo aumentou (1324 ms), e houve crescimento moderado no uso da CPU.
- O **YouTube**, por conter elementos multimídia, teve o maior tempo de carregamento (3915 ms) e aumentou significativamente o uso de CPU e RAM.
- O **recarregamento da página inicial** demonstrou o bom reaproveitamento de recursos, com redução no uso da CPU mesmo após navegação intensa.

5.2 Testes de estabilidade

Para avaliar a estabilidade do navegador em cenários reais, foram realizados testes com o carregamento de múltiplas abas contendo páginas populares e pesadas. Entre os testes realizados, foi incluída a abertura simultânea de até **10 abas** com o site do Google, o que elevou o uso de CPU e RAM até o ponto de ativar o **sistema de alerta de consumo**, projetado para evitar sobrecargas.

Além disso, foram testados sites com maior complexidade gráfica e de rede, como:

- **Rockstar Games (GTA VI)** – https://www.rockstargames.com/
- **YouTube** – https://www.youtube.com/
- **Valorant** – https://playvalorant.com/

Os resultados de tempo de carregamento e consumo de recursos foram:

Página	Tempo de Carregamento	CPU (%)	RAM (%)
Google	293 ms	7.50	67.04

Rockstar Games	977 ms	9.20	74.33
YouTube	2383 ms	4.30	73.65
Valorant	1813 ms	10.10	72.99

O navegador manteve o funcionamento estável, mas o alto consumo de recursos levou à ativação do **alerta visual de desempenho** desenvolvido no arquivo **`monitoramento.js`**, que ofereceu ao usuário a opção de encerrar o navegador caso os limites fossem ultrapassados.

5.3 Testes de rede e compatibilidade

Para avaliar a capacidade do navegador em lidar com conexões de rede e exibir corretamente páginas da web, foram realizados testes com:

- Diferentes tipos de sites (HTML puro, dinâmicos, com JavaScript pesado, vídeos embutidos).
- Protocolos **HTTP** e **HTTPS**.
- Conexões válidas e inválidas (incluindo simulações de erro de DNS e sites offline).

Compatibilidade com sites populares

O navegador foi testado em diversas páginas amplamente utilizadas, como:

Site	Tipo de Conteúdo	Resultado

Google	Busca (HTML + JS leve)	Carregou normalmente
YouTube	Vídeo em streaming, scripts pesados	Carregou normalmente
Wikipedia	Conteúdo estático (HTML/CSS)	Carregou normalmente
GTA VI - Rockstar	Animações, mídia e JS pesado	Carregou normalmente
Valorant	Design moderno e multimídia	Carregou normalmente

Todos os sites foram carregados corretamente, sem distorções visuais ou falhas de renderização. Isso mostra a **compatibilidade do motor de renderização (Chromium via Electron)** com os padrões modernos da web

5.4 Testes de segurança

A segurança é um aspecto essencial no desenvolvimento de navegadores. Para isso, foram realizados testes práticos com o objetivo de identificar possíveis vulnerabilidades, como execução de scripts maliciosos, redirecionamentos forçados e uso não autorizado de recursos sensíveis do sistema.

1. Teste de XSS (Cross-Site Scripting)

Objetivo: Verificar se o navegador permite a execução de scripts injetados via URL.

Teste com protocolo `data`:

URL usada: `data:text/html,<script>alert('XSS Vulnerável!');</script>`

Resultado: Bloqueado com erro de sintaxe (`SyntaxError`)

Conclusão: O Electron está corretamente configurado para impedir execução de código via `data:`.

Teste com protocolo `javascript:`

URL usada: `javascript:alert('XSS via protocolo!');`

Resultado: O alerta foi executado normalmente.

Conclusão: O Electron permite navegação direta com **`javascript:`**, o que pode ser explorado para ataques de XSS.

2. Teste de Open Redirect

Objetivo: Verificar se o navegador permite redirecionamentos automáticos para domínios externos via parâmetros manipulados.

- **URL usada:**
  `http://meusite.com/?redirect=https://phishing.com`
  Resultado: O navegador não seguiu o redirecionamento.
  Conclusão: O navegador está protegido contra redirecionamentos forçados.

3. Teste de Permissões de Recursos

Objetivo: Avaliar se o navegador impede o acesso não autorizado a recursos sensíveis como câmera, microfone e localização.

Câmera e microfone: `navigator.mediaDevices.getUserMedia()`

Resultado: Permissão negada.

Conclusão: Acesso bloqueado corretamente.

Geolocalização: `navigator.geolocation.getCurrentPosition()`

Resultado: Permissão negada.

Conclusão: O navegador bloqueia corretamente o acesso sem interação do usuário.

5.5 Gráficos de uso

O gráfico apresenta a variação do uso da CPU (%) ao longo do tempo durante os testes de desempenho realizados em **6 de agosto de 2022**. Ele mostra diferentes processos representados por cores distintas, com destaque para os seguintes padrões:

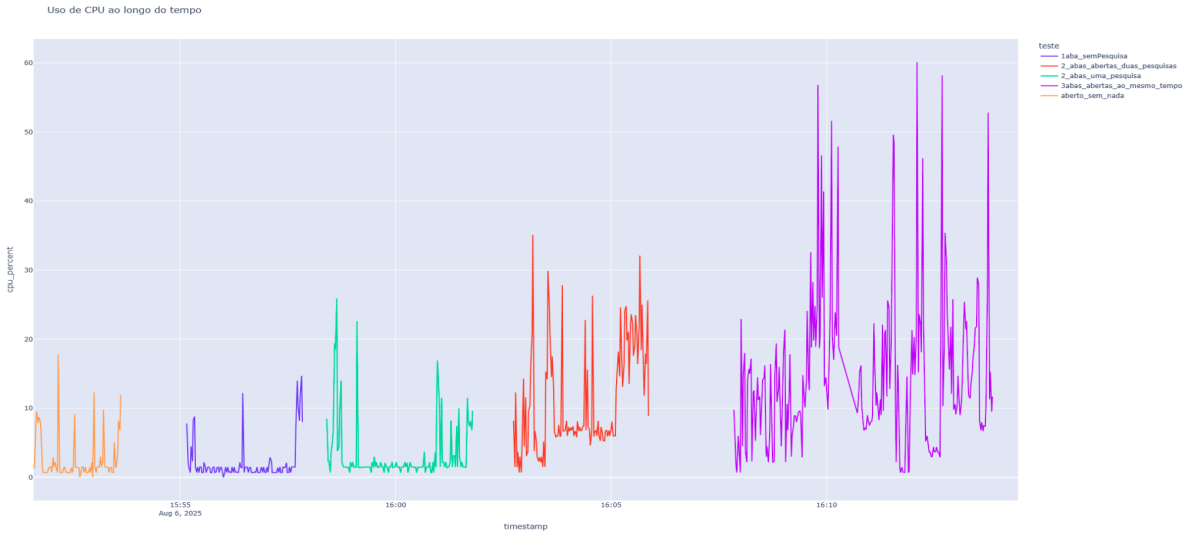
Consumo Geral de CPU

O gráfico mostra a porcentagem de uso da CPU em um sistema ao longo do tempo, no dia 6 de agosto de 2025. Cada linha colorida representa um cenário de teste diferente, conforme a legenda à direita:

- **Laranja (teste sem Pesquisa):** No início do período, por volta das 15:53, o uso da CPU oscila bastante, com picos de até cerca de 15%. No entanto, a média é bem baixa, próxima a 0%. Isso sugere que o sistema está em um estado de baixa atividade, com picos ocasionais de processamento.
- **Azul (2 abas abertas e duas pesquisas):** Entre 15:55 e 16:00, o uso da CPU é extremamente baixo e estável, com picos que não ultrapassam 10%. A maior parte do tempo, a linha fica próxima a 0%. Isso indica que, neste período, a CPU não foi muito exigida, mesmo com duas abas abertas e duas pesquisas.
- **Verde (2 abas uma pesquisa):** Após o teste azul, entre 15:58 e 16:00, o uso da CPU aumenta, com picos mais frequentes e mais altos, chegando a cerca de 25%. A média do uso da CPU neste período é visivelmente maior do que a do cenário azul, indicando que a realização de uma pesquisa em duas abas causou um aumento na atividade do processador.
- **Vermelho (2 abas uma pesquisa):** Este trecho, que ocorre entre 16:03 e 16:05, mostra um comportamento bastante agitado. O uso da CPU tem picos frequentes e elevados, chegando a cerca de 35%. Embora o nome do cenário seja o mesmo do

verde, o uso da CPU é significativamente maior, o que pode indicar que a pesquisa realizada neste momento foi mais complexa ou exigiu mais processamento, ou que o sistema estava realizando outras tarefas em segundo plano.

- **Roxo (3 abas abertas ao mesmo tempo):** Este é o cenário de maior exigência. A partir das 16:05, o uso da CPU se torna muito volátil, com picos constantes e bastante elevados. Durante este período, há diversos picos que ultrapassam 50%, e alguns chegam perto de 60%. O uso médio da CPU é o mais alto entre todos os cenários, demonstrando que a abertura de três abas simultaneamente exige muito mais do processador.



Consumo da RAM

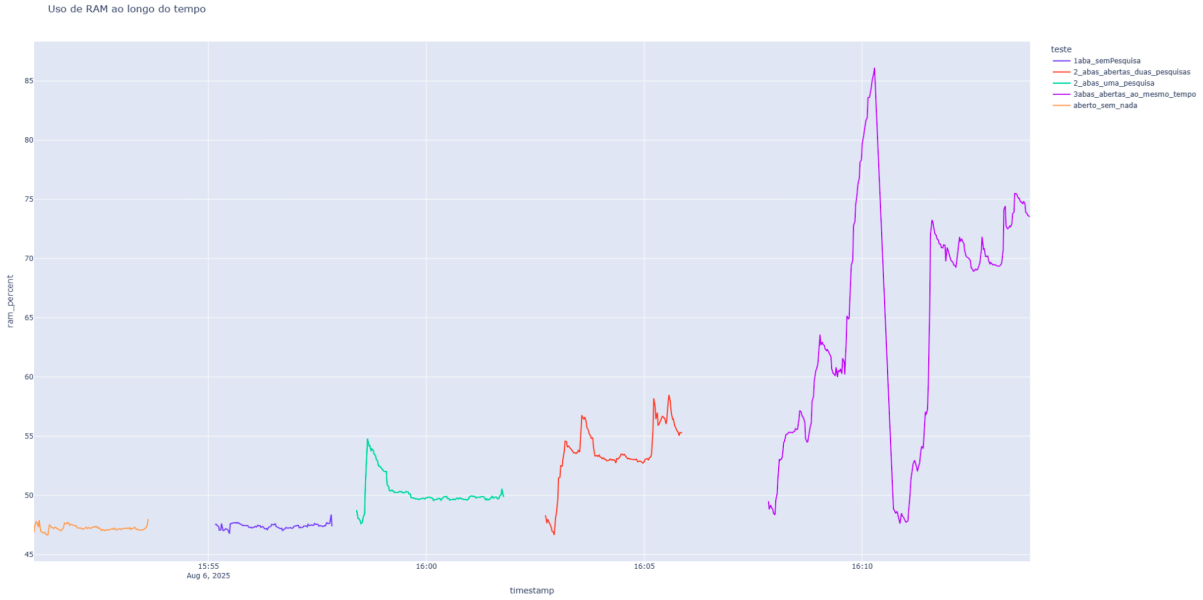
O gráfico, intitulado "Uso de RAM ao longo do tempo", mostra a porcentagem de uso da memória RAM em um sistema, ao longo de um período de tempo no dia 6 de agosto de 2025.

A legenda à direita discrimina quatro cenários de teste, representados por linhas de cores diferentes:

- **Azul (teste sem Pesquisa):** O uso da RAM é relativamente baixo e estável, oscilando entre 46% e 47% no início do período, e depois sobe ligeiramente para cerca de 47% a 48% entre 15:55 e 16:00. Este cenário parece ser uma linha de base, indicando o uso normal de RAM sem muita atividade.
- **Verde (2 abas abertas e duas pesquisas):** Após o teste azul, por volta das 15:58, o uso da RAM aumenta bruscamente de cerca de 48% para 55%, e depois se estabiliza em torno de 50%. Isso sugere que a abertura de duas abas e a realização de duas pesquisas causaram um aumento inicial no consumo de memória.
- **Laranja (2 abas e uma pesquisa):** No início do gráfico, há um trecho que pode ser um quinto cenário ou uma repetição do teste azul, onde o uso da RAM está entre 46% e 47%. No entanto, o trecho de cor laranja mais à direita, que ocorre após o teste verde, mostra o uso da RAM subindo para cerca de 55% e, posteriormente, subindo e descendo entre 53% e 57% por volta das 16:05. Isso indica que a abertura de duas abas com uma pesquisa resultou em um consumo de RAM maior do que o cenário verde, o que pode ser contra-intuitivo e mereceria uma investigação mais aprofundada.

- **Roxo (3 abas abertas ao mesmo tempo):** Este é o cenário mais notável. A partir de aproximadamente 16:05, o uso da RAM começa a subir de cerca de 48% e, por volta das 16:10, tem um aumento drástico e acelerado, atingindo o pico de cerca de 86%. Após o pico, o uso da RAM cai um pouco, mas permanece elevado, oscilando entre 68% e 75% até o final do período registrado. Este teste claramente demonstra que a abertura de três abas simultaneamente teve um impacto significativo e sustentado no consumo de memória, levando o sistema a um nível de uso muito mais alto em comparação com os outros cenários.

- **Vermelho (aberto sem nada):** este cenário mostra um uso de RAM baixo e estável. Começando em 42%, o consumo sobe e se fixa em torno de 47%. Isso pode representar o estado inicial do sistema, onde apenas o software de medição está em execução, sem nenhuma aba ou atividade adicional. A subida de 42% para 47% pode ser o resultado do carregamento completo do programa ou de algum processo em segundo plano.



6. Resultados e Discussão

O desenvolvimento do navegador web proposto demonstrou a viabilidade de se construir uma aplicação funcional, leve e com recursos relevantes, utilizando ferramentas como Electron, Node.js, JavaScript, HTML e CSS.

Durante os testes, o navegador foi capaz de:

- Renderizar corretamente sites populares e complexos.
- Gerenciar múltiplas abas com processos isolados.
- Monitorar e controlar o uso de CPU e RAM em tempo real.
- Alertar o usuário em caso de consumo elevado de recursos.
- Armazenar e exibir histórico de navegação.
- Oferecer uma interface funcional e amigável, com barra de endereços, botões de navegação e sistema de bookmarks.

- Reagir adequadamente a situações de rede instável e URLs inválidas.
- Apresentar comportamento seguro frente a acessos não autorizados e ataques simples de XSS.

Os resultados obtidos durante os testes de desempenho, rede, estabilidade e segurança indicam que o navegador atende aos requisitos propostos para um projeto acadêmico, e ainda oferece boa base para futuras melhorias.

7. Conclusão

O projeto de construção de um navegador web leve serviu como ferramenta prática para o aprofundamento de conceitos de sistemas operacionais, gerenciamento de processos, consumo de recursos e segurança em aplicações modernas.

A utilização do Electron, em conjunto com o monitoramento por terminal e a criação de componentes como `main.js`, `monitoramento.js`, `preload.js` e `seguranca.js`, permitiu uma visão clara do funcionamento interno de navegadores e de como é possível integrar a lógica de backend com a interface do usuário.

A estrutura modular do projeto, a aplicação das práticas de multiprocessamento e as estratégias de sandboxing e bloqueio de scripts maliciosos reforçaram o aprendizado em temas como segurança, concorrência e comunicação entre processos.

7.1 Considerações Finais

O navegador desenvolvido não se limita apenas à visualização de páginas web. Ele representa uma ferramenta educacional, útil para consolidar conhecimentos de desenvolvimento de software, arquitetura de aplicações, segurança da informação e integração entre sistemas.

Embora existam navegadores comerciais muito mais robustos, a proposta deste trabalho foi cumprida com sucesso ao entregar um navegador funcional, seguro e com um conjunto mínimo viável de funcionalidades esperadas.

8.Referências

ELECTRON. *Performance tutorial*. Electron Documentation. Disponível em: https://www.electronjs.org/pt/docs/latest/tutorial/performance. Acesso em: 06 ago. 2025.

COTT LOGIC. *Analysing multi-window Electron application performance using Chromium tracing*. Blog Scott Logic, 21 maio 2019. Disponível em: https://blog.scottlogic.com/2019/05/21/analysing-electron-performance-chromium-tracing.html. Acesso em: 06 ago. 2025.

BRAINHUB. *Electron App Performance – How to Optimize It*. Brainhub, jul. 2024. Disponível em: https://brainhub.eu/library/electron-app-performance. Acesso em: 06 ago. 2025.

PALETTE. *6 Ways Slack, Notion, and VSCode Improved Electron App Performance*. Palette Blog, maio 2025. Disponível em: https://palette.dev/blog/improving-performance-of-electron-apps. Acesso em: 06 ago. 2025.

GITHUB. *Issue: Electron memory leak when using multiple listeners*. Electron GitHub Repository, 2021. Disponível em: https://github.com/electron/electron/issues/31770. Acesso em: 06 ago. 2025.

YOUTUBE. *Criando um navegador com Electron*. [S.l.]: YouTube, 2023. 1 vídeo (11min). Publicado por "Canal Programador BR". Disponível em: https://www.youtube.com/watch?v=P1gQ7Ppd48U. Acesso em: 06 ago. 2025.

YOUTUBE. *Como criar um navegador com Electron passo a passo*. [S.l.]: YouTube, 2022. 1 vídeo (14min). Publicado por "Pedro Tech". Disponível em: https://www.youtube.com/watch?v=zy-B9d2ZVrw. Acesso em: 06 ago. 2025.