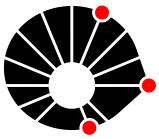

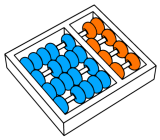

Migração de um Sistema Distribuído para WebAssembly

D. M. de Morais *G. de L. Palma* *L. F. Bittencourt*
R. R. Filho *A. R. B. P. Barata*

Relatório Técnico - IC-PFG-25-35
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Migração de um Sistema Distribuído para WebAssembly

Daniela Marques de Morais* Gustavo de Lima Palma†
Luiz Fernando Bittencourt‡ Roberto Rodrigues Filho§
Arthur Rezende Bueno Pontes Barata¶

Resumo

O documento descreve a migração de um sistema de multiplicação de matrizes, antes totalmente nativo em Dana, para uma arquitetura distribuída baseada em WebAssembly executando no navegador. A mudança foi motivada pelas limitações do ambiente web, que impede o uso de sockets TCP e operações bloqueantes. Isso exigiu a troca do RPC por uma API REST, a simplificação do mecanismo de adaptação dinâmica e a adoção de um coordenador nativo responsável pela fila de tarefas, distribuição do trabalho e coleta de resultados.

A nova arquitetura combina três elementos: a aplicação principal em Wasm, que envia matrizes e realiza polling pelos resultados; workers também em Wasm, cada um executado em uma aba do navegador; e o coordenador nativo, que mantém o estado global e garante sincronização com mutexes. Todo o fluxo é assíncrono para evitar travamentos no navegador. Diversos ajustes foram necessários durante o desenvolvimento, como substituir chamadas HTTP bloqueantes por versões assíncronas, padronizar a serialização em JSON e delegar o gerenciamento de sockets ao módulo ws.core para evitar respostas truncadas.

O resultado é um sistema funcional capaz de distribuir a multiplicação de matrizes entre vários workers Wasm, retornando o resultado à aplicação principal. Ainda há oportunidades claras de avanço, como análise de desempenho, melhoria da robustez do coordenador e expansão da arquitetura para múltiplas instâncias, mas a viabilidade da abordagem foi demonstrada.

1 Introdução

Sistemas autodistribuídos (Self-Distributing Systems ou SDS) são sistemas que permitem seu próprio software determine de forma autônoma a sua arquitetura mais apropriada em tempo real, com base nas condições operacionais atuais. Ao invés de ter engenheiros envolvidos na tomada de decisão sobre a distribuição do sistema, os SDS delegam as escolhas

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
‡Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
§Departamento de Ciência da Computação, Universidade de Brasília, 70910-900 Brasília, DF.
¶Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

de design distribuído inteiramente a processos autônomos. [1] Ao realocar ou replicar componentes em tempo de execução, os SDS permitem otimizar a performance, movendo a computação para mais perto de recursos poderosos na nuvem quando necessário.

A linguagem de programação Dana foi desenhada especificamente para suportar este paradigma. Como uma linguagem orientada a componentes, Dana é construída sobre componentes de software onde tipos não primitivos são considerados passíveis de hot-swap. O seu runtime suporta a adaptação transparente do software, substituindo um componente por uma das suas variantes em tempo de execução e de forma segura [2]. Esta capacidade é viabilizada pelo implementação da mesma interface entre o proxy e o componente equivalente, que permite a troca entre eles de forma transparente.

O sistema de multiplicação de matrizes (MatMul) serve como um estudo de caso para estes conceitos [3]. Originalmente, o MatMul foi implementado como um sistema puramente nativo em Dana: O sistema inicia como uma aplicação local e, ao detectar um aumento na carga, substitui dinamicamente o seu componente de multiplicação de matrizes por um proxy que distribui o trabalho para workers remotos através de RPC. No entanto, o Dana ainda possui limitações relacionadas à sua implementação em larga escala. Por isso, a compilação de um projeto tão essencial como MatMul para Wasm, uma linguagem já muito difundida, torna a implementação de soluções reais baseadas em Dana mais palpáveis.

O foco do projeto é a **migração e reformulação deste sistema para um ambiente de computação distribuída baseado em WebAssembly (Wasm)**, explorando o navegador web como uma plataforma de processamento.

2 Motivação e limitações

A adoção do WebAssembly nesta migração oferece várias vantagens estratégicas para além da execução no navegador:

- **Portabilidade**: O WebAssembly fornece um formato binário do tipo "escreva uma vez, execute em qualquer lugar". Isto simplifica a distribuição de componentes, já que um único binário Wasm pode ser executado em qualquer arquitetura, incluindo navegadores, ao contrário da compilação nativa.
- **Independência da Linguagem**: A migração para Wasm desacopla a arquitetura (SDS) da linguagem de programação. A lógica de alto desempenho, como a multiplicação de matrizes, pode ser implementada em Rust ou C++ para otimizações de baixo nível, enquanto a lógica de orquestração pode ser escrita numa linguagem de mais alto nível, com todos os componentes a compilar para um formato Wasm interoperável.

Em contrapartida, a computação distribuída em ambientes web apresenta desafios únicos devido às restrições de segurança dos navegadores, que limitam o acesso direto a sockets TCP e operações de rede de baixo nível. WebAssembly (Wasm) oferece uma solução promissora mas requer padrões de design específicos para manter a responsividade da interface do utilizador.

Esta transição implicou numa reformulação fundamental do sistema:

- **Comunicação**: O modelo de RPC sobre TCP foi substituído por uma API RESTful
- **Simplificação da Arquitetura**: O sistema de adaptação dinâmica e autônoma foi substituído por uma arquitetura distribuída mais simples com um coordenador nativo e workers em Wasm. A adaptação dinâmica do sistema original foi temporariamente simplificada para focar na resolução dos desafios fundamentais da plataforma Wasm. Esta decisão permitiu estabelecer a viabilidade técnica da abordagem antes de reintroduzir complexidade adicional.
- **Ambiente de Execução**: O sistema evoluiu de um serviço de backend puramente nativo para um sistema híbrido: a interface de usuário e os workers executando no navegador (Wasm), e um coordenador nativo para gerir tarefas.
- **Separação de Responsabilidades do Coordenador**: A lógica central de coordenação foi implementada em `server/CoordinatorController.dn`, que gerencia o ciclo de vida das tarefas. A exposição dessa lógica via HTTP é feita por `ws/CoordinatorWeb.dn`, um adaptador que se integra ao servidor web, tratando de requisições e cabeçalhos (CORS), e que carrega dinamicamente o controlador principal.

3 Evolução da Arquitetura

A arquitetura do sistema puramente em Dana nativo é organizada em torno de um componente principal `Server` que atua como um servidor HTTP e orquestra a lógica de adaptação em tempo de execução. O ponto de entrada da aplicação, `Main`, permite iniciar o servidor em três modos distintos: modo local, modo distribuído utilizando um proxy desde o início e modo adaptativo que alterna entre local e distribuído com base na performance. Para a computação distribuída, existe o componente `Worker` que escuta numa porta TCP para receber requisições de cálculo via RPC. As requisições HTTP para o endpoint `/matmul` eram recebidas e o cálculo é delegado ao componente `Matmul`, o qual contém a lógica central para a multiplicação de matrizes. Para viabilizar a distribuição, `MatmulProxy` divide o trabalho e o distribui para os workers remotos, gerenciando as chamadas RPC. O componente `Server` monitora os tempos de resposta e utiliza esse proxy para alternar dinamicamente para o modo de execução distribuída quando a performance da computação local se degradava, realizando a adaptação em tempo de execução.

A definição da arquitetura final em WebAssembly foi um processo iterativo, a estrutura final do sistema emergiu da resolução sucessiva de erros de compilação e de execução e é composto por três tipos principais de componentes:

1. **Aplicação Principal (Wasm)**: A interface de usuário que submete tarefas e exibe resultados, executando no navegador (`app/main.dn` + `app/MainAppLoopImpl.dn`).
2. **Workers (Wasm)**: Os processadores distribuídos que executam as multiplicações de matrizes, cada um executando em uma aba separada do navegador (`app/BrowserWorkerWasm.dn` + `app/BrowserWorkerLoopImpl.dn`).

3.1 Diagrama de Arquitetura

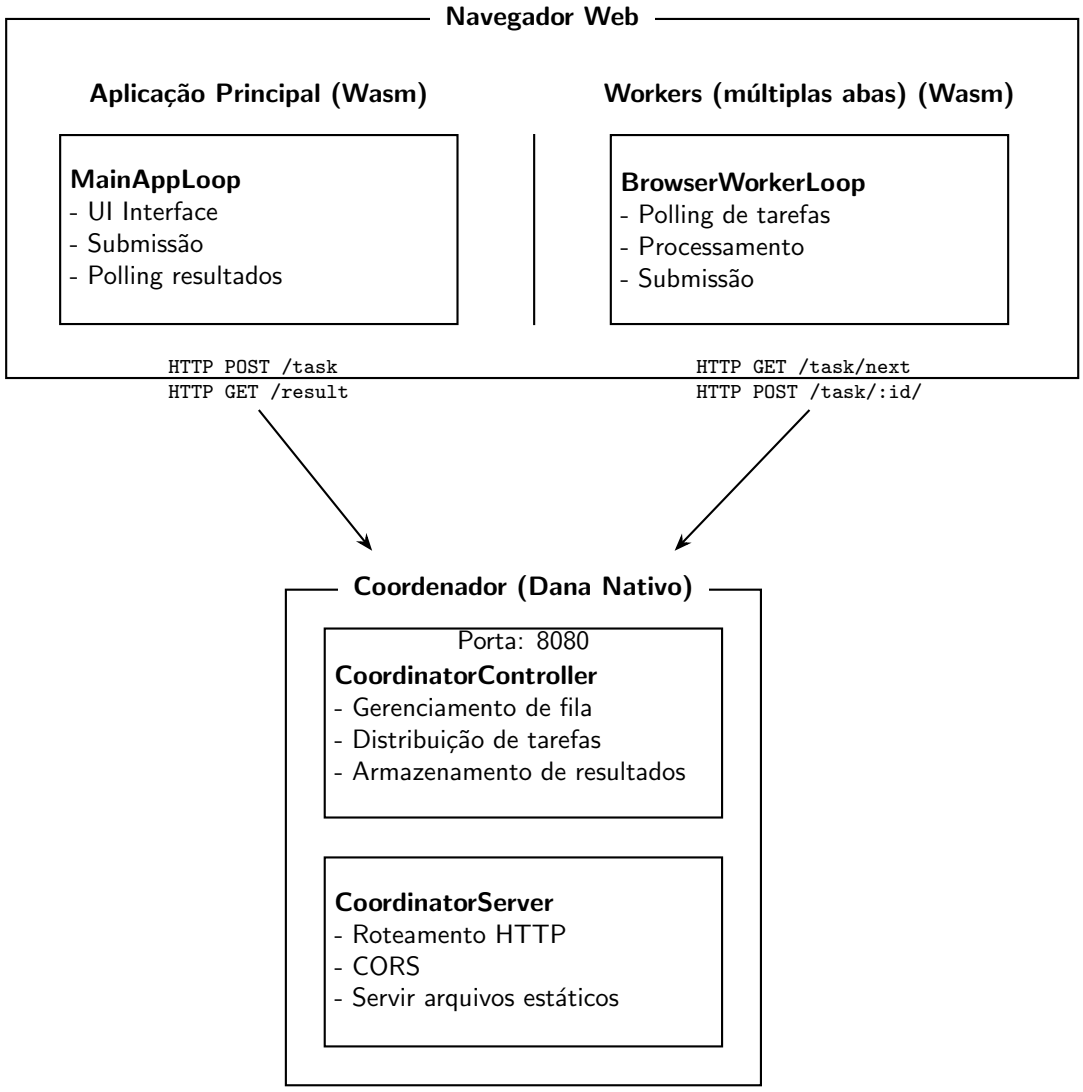

Figura 1: Diagrama da Arquitetura do Sistema.

3.2 Diagrama de Sequência

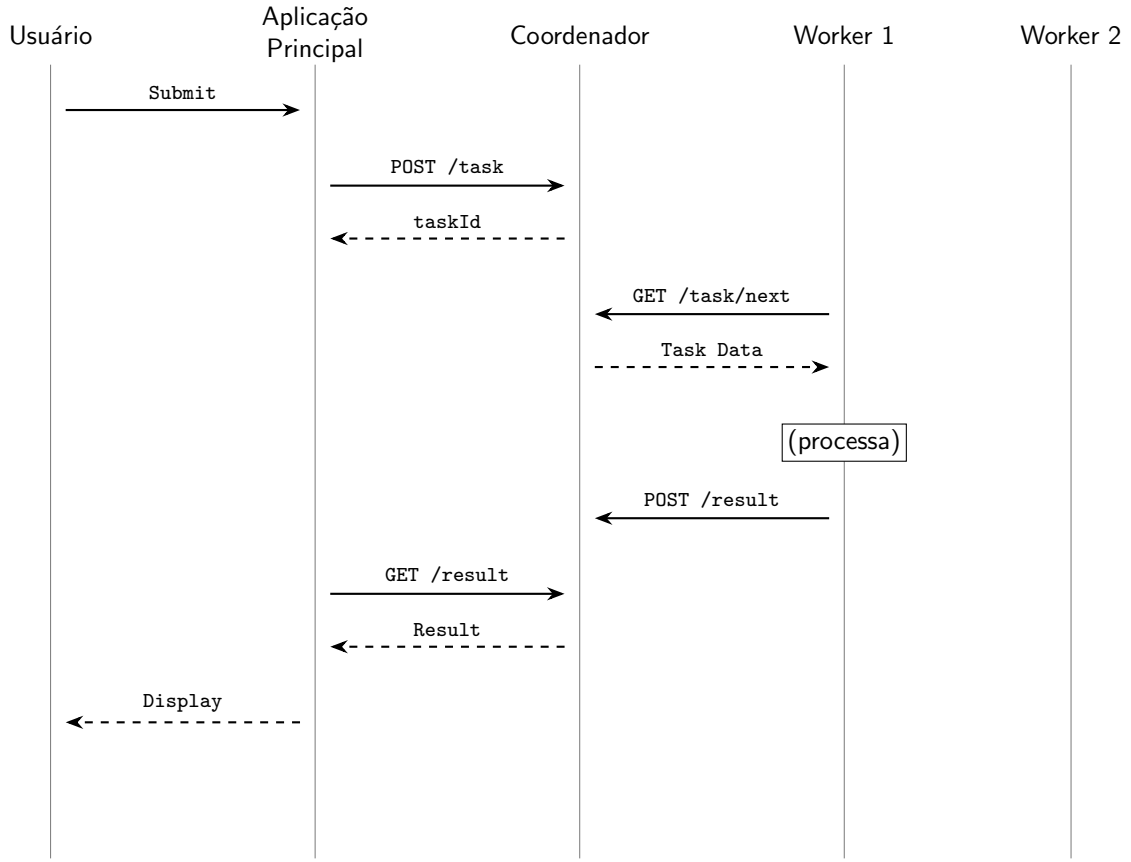

Figura 2: Diagrama de Sequência da Multiplicação de Matrizes.

3.3 Componentes na nova arquitetura

A aplicação principal (`app/main.dn`) fornece a interface de utilizador para submissão das matrizes: envia as matrizes em formato JSON para o coordenador via `POST /task`, recebe um ID de tarefa e, de seguida, faz polling periódico por resultados através de `GET /result/:id`. Os workers (`app/BrowserWorkerWasm.dn`) operam em abas de navegador independentes, solicitando tarefas ao coordenador com `GET /task/next`. Realizam a multiplicação de matrizes utilizando o componente `matmul/Matmul.dn` e submetem o resultado de volta ao coordenador via `POST /task/:id/result`.

Ambos são executados em WebAssembly e utilizam o padrão `ProcessLoop` de Dana para operações não-bloqueantes, essencial para manter a responsividade da interface do utilizador e todas as requisições HTTP são executadas de forma assíncrona.

3.3.1 Servidor Coordenador em Dana Nativo

O coordenador é executado como uma aplicação Dana nativa para possibilitar a comunicação entre os workers e a aplicação principal, é necessário pois o ambiente WebAssembly não permite operações de rede de baixo nível como a vinculação a portas TCP. A lógica reside no componente `server/CoordinatorController.dn`: gere uma fila de tarefas em memória e expõe uma API REST para receber novas tarefas da aplicação principal, distribuir tarefas pendentes para os workers, receber os resultados dos workers e fornecer os resultados finais à aplicação principal.

As operações que modificam o estado compartilhado são protegidas por mutexes para garantir a segurança em ambientes com concorrência (*thread safety*), e todas as respostas HTTP incluem os cabeçalhos CORS necessários para a comunicação com o navegador.

3.4 Fluxo de Dados

O fluxo completo de uma multiplicação de matrizes funciona da seguinte forma:

1. **Submissão de Tarefa**:
   - Usuário abre a aplicação principal em uma aba do navegador (http://localhost:8081/xdana.html)
   - A aplicação carrega o runtime Dana Wasm e o arquivo `file_system.js` (contém todos os componentes compilados empacotados)
   - Usuário insere duas matrizes e clica em "Submit"
   - Aplicação constrói requisição HTTP POST para http://localhost:8080/task com corpo JSON contendo as matrizes A e B
   - Requisição é executada de forma assíncrona usando `asynch::executeSubmitRequest`
2. **Processamento pelo Coordenador**:
   - Coordenador recebe a requisição, cria nova tarefa com ID único
   - Armazena dados da tarefa em memória
   - Adiciona ID à fila de tarefas pendentes
   - Retorna resposta JSON com `taskId`
3. **Polling pela Aplicação Principal**:
   - Aplicação principal recebe `taskId` e entra em estado de polling
   - Faz requisições GET periódicas para `/result/:id` até que o resultado esteja disponível
4. **Processamento pelo Worker**:
   - Worker(s) executando em abas separadas fazem polling no endpoint `/task/next?workerId=X`
   - Quando há tarefa pendente, coordenador:

  - Remove tarefa da fila
  - Marca status como "processing"
  - Atribui worker ID à tarefa
  - Retorna dados das matrizes A e B
- Worker recebe tarefa, converte JSON para `Matrix`, realiza multiplicação, converte resultado de volta para JSON
- Worker submete resultado ao coordenador via POST `/task/:id/result`

5. **Retorno do Resultado**:
   - Coordenador recebe resultado, atualiza status para "completed", armazena resultado
   - Quando aplicação principal faz próximo poll em `/result/:id`, coordenador retorna resultado completo
   - Aplicação principal exibe resultado na interface do usuário

4 Processo de decisão da arquitetura

Durante o desenvolvimento, foram encontrados diversos problemas de compilação ou durante a execução, que estão, juntos de suas soluções, representados pelos itens abaixo:

4.1 O Padrão ProcessLoop e Operações Não-Bloqueantes

Qualquer operação bloqueante dentro do `loop()` causa o travamento do navegador.

Solução: Operações potencialmente demoradas devem ser movidas para *threads* assíncronas (usando `asynch::`) ou reestruturadas para usar padrões não-bloqueantes, como o *polling* com contadores para gerir atrasos e verificar o estado de operações.

Exemplo de implementação de polling não-bloqueante:

```
int pollCounter = 0
const int POLL_INTERVAL_LOOPS = 50  // ~500ms

bool loop() {
    if (state == POLLING) {
        pollCounter++
        if (pollCounter >= POLL_INTERVAL_LOOPS) {
            pollCounter = 0
            asynch::executePollRequest(taskId)
        }
    }
    return true
}
```

4.2 Requisições HTTP em Wasm

`net.http.HTTPRequest` é uma operação bloqueante e por isso não pode ser usada diretamente dentro do método `loop()` do ProcessLoop em Wasm.

Solução: Criar funções separadas (ex: `executeSubmitRequest`, `executePollRequest`) que são chamadas com `asynch::`, e usar flags de estado como `waitingForResponse` para verificar no `loop()` quando a resposta está disponível:

```
bool waitingForResponse = false
HTTPResponse currentResponse = null

bool loop() {
    if (waitingForResponse) {
        if (currentResponse != null) {
            handleResponse(currentResponse)
            waitingForResponse = false
            currentResponse = null
        }
        return true
    }
    // ... resto da lógica
}

void executeSubmitRequest(char url[], Header headers[], char postData[]) {
    waitingForResponse = true
    currentResponse = http.post(url, headers, postData, false)
}
```

4.3 Buffers de Socket TCP e Transmissão de Dados

A implementação inicial do servidor, que utilizava `net.TCPServerSocket` enfrentava problemas de transmissão incompleta de dados. As respostas HTTP eram por vezes truncadas porque a conexão do *socket* era fechada antes do *buffer* de envio do sistema operativo ser completamente transmitido.

Solução: Ao invés de implementar uma solução manual de *flushing* de *buffer*, o problema foi resolvido de forma mais robusta ao migrar o servidor para usar `ws.core`. A implementação em `ws/CoordinatorWeb.dn` delega todo o gerenciamento de *sockets* e a transmissão de respostas ao `ws.core` que garante a entrega completa dos dados, eliminando a necessidade de manipulação direta de *sockets*.

4.4 Serialização JSON

Tivemos inconsistências em como os dados JSON eram estruturados em diferentes pontos do sistema.

Solução: Padronizar formato: matrizes são sempre enviadas como strings JSON (ex: `"[[1,2],[3,4]]"`), usando `data.json.JSONEncoder` e `data.json.JSONParser` consistentemente.

4.5 Race Conditions no Coordenador

Múltiplos workers fazendo polling simultaneamente poderiam receber a mesma tarefa.

Solução: Garantir que remoção da fila e atualização do status aconteçam atomicamente dentro do mesmo bloco mutex, verificando status da tarefa antes de atribuí-la a um worker:

```
mutex(lock) {
    if (taskQueue.arrayLength > 0) {
        int taskId = taskQueue[0]
        Task task = findTask(taskId)
        if (task != null && task.status == "pending") {
            taskQueue = removeFromQueue(taskQueue, 0)  // Remove atomicamente
            task.status = "processing"        // Atualiza status atomicamente
            task.workerId = workerId
            return task
        }
    }
}
```

5 Testes

A validação do sistema concentrou-se em testes funcionais para garantir a corretude da multiplicação de matrizes e do fluxo completo de execução.

1. **Testes Funcionais Manuais**: Foram realizados para validar o fluxo completo do sistema, desde a submissão de uma tarefa de multiplicação de matriz até o recebimento do resultado. Estes testes verificaram a interação entre o coordenador e os *workers* no cenário distribuído.
2. **Testes Automatizados (Coordenador)**: Foram implementados testes automatizados (test-coordinator.sh) focando na lógica de gestão de tarefas e na corretude dos cálculos em modo de execução local.

Os critérios de sucesso para a validação funcional foram:

- Executa multiplicações de matrizes corretamente, tanto no modo local como no distribuído.
- O coordenador distribui a carga de trabalho entre os *workers* disponíveis.
- A aplicação cliente recebe o resultado correto da multiplicação.
- O sistema gere a fila de tarefas de forma correta, processando os pedidos por ordem de chegada.

5.1 Trabalhos Futuros

Uma área crítica para trabalhos futuros é a introdução de uma análise de desempenho quantitativa abrangente como: entender se há overhead considerável adicionado pela comunicação baseada em HTTP em comparação com o sistema original, medir como o tempo de resposta varia à medida que o número de *workers* Wasm aumenta e uma comparação de desempenho direta entre a versão nativa original e a nova arquitetura baseada em Wasm.

Além disso, há a necessidade de testar problemas de concorrência no código do servidor, já que foram observados alguns bugs demonstrados pela falta de resposta do servidor quando múltiplas requisições são feitas em um curto espaço de tempo.

No que diz respeito à robustez e escalabilidade, seria crucial introduzir múltiplos coordenadores com um balanceador de carga para eliminar o ponto único de falha, bem como implementar timeouts e um sistema de descoberta automática para os *workers*. O aprofundamento da análise e dos testes sobre a adaptação dinâmica entre as implementações locais e distribuídas continua a ser uma área central para investigação futura. Além disso, a experiência do utilizador poderia ser significativamente melhorada através de aprimoramentos na interface e da implementação de uma funcionalidade para visualizar o progresso das tarefas em tempo real.

6 Considerações finais

O sistema implementado demonstra uma arquitetura funcional de multiplicação de matrizes distribuída usando Dana, com componentes principais e workers executando em Wasm e um coordenador executando nativamente. A implementação resolveu vários desafios técnicos relacionados a Wasm, ProcessLoops, comunicação HTTP e gerenciamento de estado.

Apesar de algumas melhorias poderem ser necessárias para explorar a fundo, os resultados demonstram que a abordagem é viável, é adequado para demonstração e abre caminhos para futuras investigações.

Referências

[1] R. Rodrigues-Filho and B. Porter, *Hatch: Self-distributing systems for data centers, Future Generation Computer Systems,* **132**, p. 80 (2022).

[2] R. Rodrigues Filho, B. Porter, F. M. Costa and I. Sene Júnior, *Emergent Software Systems: Theory and Practice,* Livro-texto de Minicursos XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos — SBRC (2021)

[3] A. Barata, R. Rodrigues-Filho, C. A. Astudillo, and L. F. Bittencourt, *An Interface Definition Language for Supporting Stub Generation in Self-Distributing Systems, 2025 IEEE International Conference on Quantum Software (QSW)*, 2025, pp. 269–274, doi: 10.1109/QSW67625.2025.00040.