Relatório Técnico – Orquestração de Tarefas

Nomes: **Maria Julia de Carvalho Costa**

Laura Venancio Martins

Sara Ramos Scalia

RA's: Maria Júlia - 22408994

Laura Venâncio - 22407774

Sara Ramos - 22401389

Introdução

O projeto prático tem como objetivo simular a execução e orquestração de tarefas em um cluster de servidores de inferência. Cada requisição de inteligência artificial, seja de visão computacional, processamento de linguagem natural ou de voz, é atribuída a um servidor com capacidade limitada. A distribuição dessas tarefas é gerenciada por um orquestrador, que aplica políticas de escalonamento de forma a otimizar o desempenho do sistema e balancear a carga entre os servidores.

O foco do projeto é analisar o comportamento do sistema, verificando métricas como tempo de resposta médio, throughput e utilização simulada de CPU, além de permitir a migração dinâmica de tarefas entre servidores.

Arquitetura do Sistema

A arquitetura do projeto segue um modelo modular, organizada em diretórios específicos para cada componente do sistema.

```
sistemas-operacionais-/
│
├── src/
│   ├── __init__.py
│   │
│   ├── orchestrator/
│   │   ├── __init__.py
│   │   ├── orchestrator.py
│   │   ├── scheduler.py
│   │   └── metrics.py
│   │
│   ├── servers/
│   │   ├── __init__.py
│   │   └── worker.py
│   │
│   └── utils/
│       ├── __init__.py
│       └── logger.py
│
├── tests/
│   ├── test_scheduler.py
│   ├── test_orchestrator.py
│   └── test_worker.py
│
├── docs/
│   ├── arquitetura.txt
│   ├── Diagrama2.pdf
│   └── relatório.pdf
│
├── main.py
├── input.json
├── requirements.txt
├── run.bat
└── pytest.ini
```

O diretório **src** contém o código-fonte, organizado em módulos: **orchestrator**, responsável pela criação do orquestrador e escalonamento de tarefas; **servers**, com os workers que processam as tarefas; e **utils**, contendo ferramentas de suporte, como logging.

O arquivo **main.py** na raiz inicializa o orquestrador com as requisições definidas em **input.json**. Os testes unitários estão na pasta **tests** e arquivos auxiliares, como

requirements.txt e **run.sh**, ficam na raiz para facilitar execução e configuração do ambiente.

Entrada de Dados

As requisições e servidores são definidos em um arquivo JSON de entrada, como segue:


```
{
  "policy": "RR",
  "servidores": [
     { "id": 1, "capacidade": 3 },
     { "id": 2, "capacidade": 2 },
     { "id": 3, "capacidade": 1 },
     { "id": 4, "capacidade": 4 },
     { "id": 5, "capacidade": 2 },
     { "id": 6, "capacidade": 3 }
  ],
  "requisicoes": [
     { "id": 1, "tipo": "vc", "prioridade": 1, "tempo_exec": 2 },
     { "id": 2, "tipo": "nlp", "prioridade": 2, "tempo_exec": 4 },
     { "id": 3, "tipo": "voz", "prioridade": 1, "tempo_exec": 3 },
     { "id": 4, "tipo": "vc", "prioridade": 3, "tempo_exec": 5 },
     { "id": 5, "tipo": "nlp", "prioridade": 1, "tempo_exec": 1 },
     { "id": 6, "tipo": "voz", "prioridade": 2, "tempo_exec": 2 },
     { "id": 7, "tipo": "vc", "prioridade": 3, "tempo_exec": 4 },
     { "id": 8, "tipo": "nlp", "prioridade": 2, "tempo_exec": 1 },
     { "id": 9, "tipo": "voz", "prioridade": 1, "tempo_exec": 3 },
     { "id": 10, "tipo": "vc", "prioridade": 1, "tempo_exec": 2 },
```

```
    { "id": 11, "tipo": "nlp", "prioridade": 3, "tempo_exec": 5 },
    { "id": 12, "tipo": "voz", "prioridade": 3, "tempo_exec": 4 },
    { "id": 13, "tipo": "vc", "prioridade": 2, "tempo_exec": 3 },
    { "id": 14, "tipo": "nlp", "prioridade": 1, "tempo_exec": 2 },
    { "id": 15, "tipo": "voz", "prioridade": 2, "tempo_exec": 5 },
    { "id": 16, "tipo": "vc", "prioridade": 1, "tempo_exec": 1 },
    { "id": 17, "tipo": "nlp", "prioridade": 2, "tempo_exec": 4 },
    { "id": 18, "tipo": "voz", "prioridade": 3, "tempo_exec": 3 },
    { "id": 19, "tipo": "vc", "prioridade": 1, "tempo_exec": 1 },
    { "id": 20, "tipo": "nlp", "prioridade": 2, "tempo_exec": 2 }
  ]
}
```


Seis servidores possuem capacidades diferentes e são atribuídas vinte requisições de diferentes tipos e prioridades. A política de escalonamento utilizada é **Round Robin**, que distribui as tarefas de forma circular entre os servidores.

Funcionamento do Sistema

O orquestrador inicializa os workers correspondentes aos servidores e distribui as requisições de acordo com a política definida. Cada worker processa suas tarefas simultaneamente, respeitando sua capacidade máxima. O sistema registra logs detalhados de cada atribuição e conclusão de tarefas, permitindo monitorar em tempo real o fluxo de execução.

Quando um servidor atinge sua capacidade máxima, tarefas podem ser migradas para outros servidores disponíveis, garantindo balanceamento de carga e evitando atrasos. Ao final, o orquestrador calcula métricas de desempenho, como tempo médio de resposta e throughput.

Saída Observada:

A execução do sistema gerou os seguintes logs:

[00:00] Orquestrador iniciado
[00:00] Requisição 1 atribuída ao Servidor 1
[00:00] Requisição 2 atribuída ao Servidor 2
[00:00] Requisição 3 atribuída ao Servidor 3
...
[00:03] Tarefa 15 migrada do Servidor 3 para 2
[00:03] Servidor 3 concluiu tarefa 3
[00:05] Servidor 2 concluiu tarefa 15
[00:11] Orquestrador finalizando...
[00:11] Todos os workers finalizados
--- Resumo de métricas ---
Tarefas concluídas: 20
Tempo médio de resposta: 1.78s
Throughput (últimos 10s): 1.40 tasks/s

Contudo, as tarefas foram distribuídas de forma circular entre os servidores, com migrações ocorrendo quando necessário para manter o balanceamento de carga. Todas as tarefas foram concluídas, e o sistema apresentou desempenho estável.

Análise Comparativa das Políticas de Escalonamento

Comparando as diferentes políticas de escalonamento:

- Round Robin (RR): Distribui as tarefas de forma circular entre os servidores, garantindo justiça, mas sem priorizar tarefas mais curtas ou críticas.. É simples de implementar e adequado para cenários de carga balanceada.

- Shortest Job First (SJF): Prioriza tarefas com menor duração, reduzindo o tempo médio de resposta, mas podendo causar starvation (atrasar) de tarefas maiores.

- Prioridade: Executa primeiro as tarefas com maior prioridade, atendendo rapidamente tarefas críticas, porém atrasando tarefas de baixa prioridade.

A política RR utilizada neste teste garantiu que todas as requisições fossem distribuídas de forma equilibrada entre os servidores, sem complexidade adicional.

Conclusão

O projeto demonstra a aplicação de conceitos de programação concorrente e escalonamento de tarefas em sistemas distribuídos. A arquitetura modular facilita manutenção e expansão, permitindo novas políticas de escalonamento ou tipos de requisição. A coleta de métricas e logs detalhados fornece informações essenciais para análise do desempenho do cluster e comportamento do sistema em diferentes cenários.

O sistema garante processamento eficiente, suporte a migração dinâmica de tarefas e fornece base sólida para futuras melhorias e testes comparativos de políticas de escalonamento

.