Universidade Federal de Uberlândia - UFU
Faculdade de Computação - FACOM
Bacharelado em Sistemas de Informação - Campus Monte Carmelo
Disciplina: Sistemas Distribuídos

Nome: Lucas Dornelles
Matrícula: 31811BSI026

Sistema de Gerenciamento de Tarefas Distribuído (To-Do List)

Descrição do Tema

O projeto consiste no desenvolvimento de um **Sistema de Gerenciamento de Tarefas Distribuído (To-Do List)**, onde múltiplos clientes podem interagir com uma lista compartilhada de tarefas. O sistema possui:

- Um **Servidor Central (Líder)** que coordena as operações
- **Nós Secundários** (réplicas) que mantêm cópias sincronizadas da lista

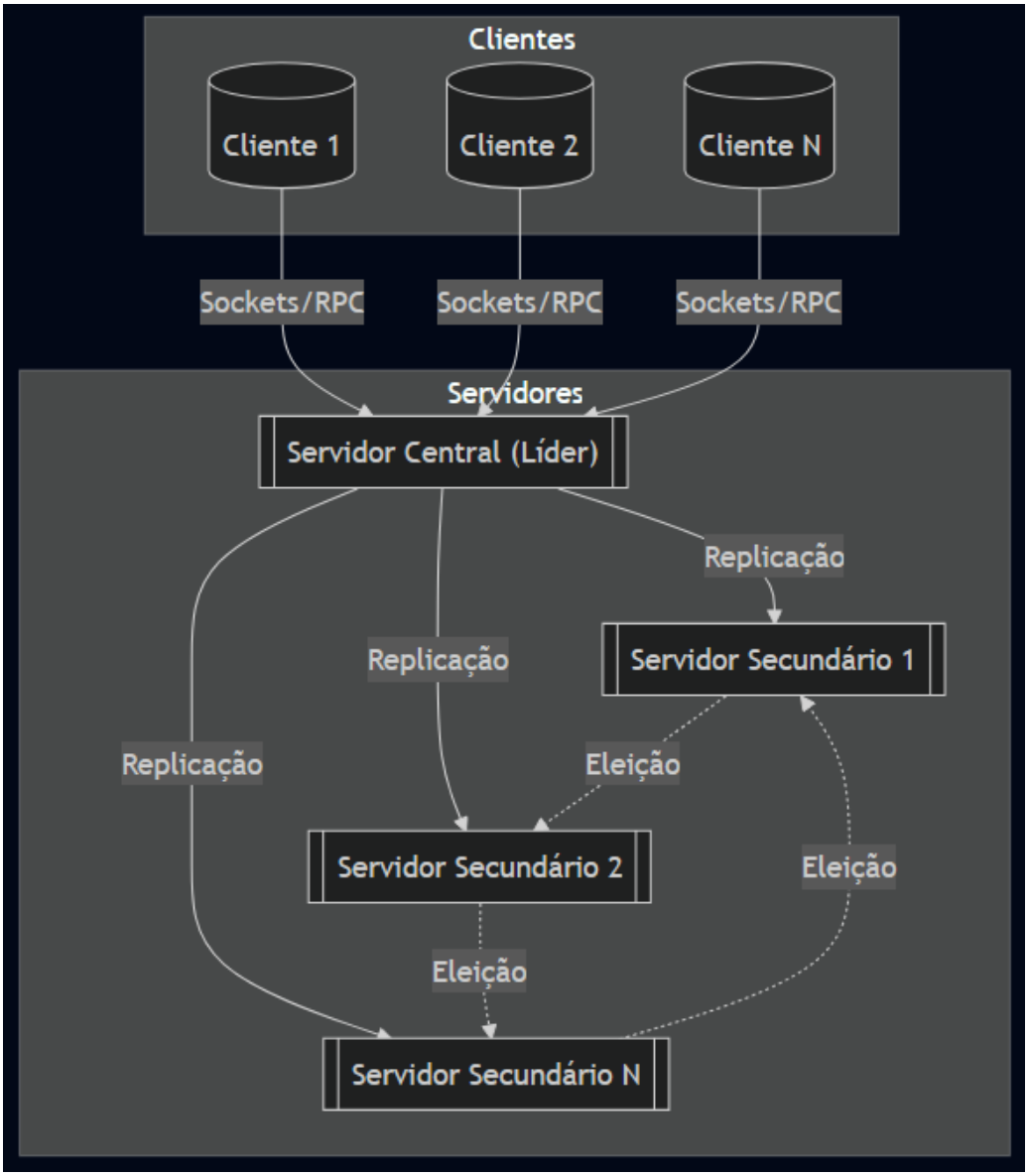
Fluxo Principal:

1. Cliente envia requisição (add/remove/edit/list) ao líder via **Sockets/RPC**
2. Líder processa e replica a mudança para os nós secundários
3. Se o líder falhar, os nós iniciam um **protocolo de eleição**

Metas de Sistemas Distribuídos

Meta	**Implementação**
Escalabilidade	Adição dinâmica de nós secundários
Disponibilidade	Failover automático para réplicas
Tolerância a Falhas	Eleição de líder + heartbeat
Consistência	Modelo eventualmente consistente
Exclusão Mútua	Timestamps para conflitos

Diagrama da Arquitetura



Componentes

Cliente: Processo que envia operações via Sockets/RPC. Interface pode ser CLI ou GUI.

Servidor Central (Líder): Processo principal que gerencia a lista global, coordena réplicas e implementa lógica de eleição.

Servidores Secundários: Processos que mantêm réplicas atualizadas e participam da eleição.

Etapa 2: Implementação com Threads e Comunicação via Sockets

Descrição do Estado Atual do Projeto

Nesta etapa, o projeto está funcional com três componentes principais: servidor líder, servidores secundários e clientes. Os clientes são capazes de se conectar simultaneamente ao servidor líder e enviar comandos para adicionar, remover, editar e listar tarefas. O líder processa essas operações e replica as mudanças para os servidores secundários via comunicação socket. Cada secundário mantém uma cópia atualizada da lista de tarefas.

Justificativa: Processos vs. Threads

A implementação foi feita utilizando **threads**, em vez de processos, pelos seguintes motivos:

- **Baixo custo de criação e gerenciamento**: Threads são mais leves e rápidas para serem criadas.
- **Compartilhamento de memória**: As threads compartilham o mesmo espaço de memória, o que facilita o acesso e atualização da lista de tarefas entre diferentes conexões simultâneas.
- **Sincronização simplificada**: Com o uso de locks, é possível garantir exclusão mútua de forma simples e eficiente.

Descrição da Comunicação via Sockets

Toda a comunicação entre os componentes do sistema ocorre via **Sockets TCP**. A escolha do protocolo TCP se deu por garantir:

- **Confiabilidade na entrega dos dados**
- **Manutenção da ordem das mensagens**
- **Correção de erros automática no nível de transporte**

Conexões implementadas:

- **Clientes <−> Servidor Líder**: os clientes enviam comandos (add, remove, edit, list) e recebem respostas.
- **Servidor Líder <−> Servidores Secundários**: o líder replica alterações para as réplicas sempre que a lista é modificada.

Formato das mensagens:

- `add;Tarefa`
- `remove;Tarefa`
- `edit;Antiga;Nova`
- `list`

Instruções de Execução

1. Certifique-se de ter o Python 3 instalado no sistema.
2. No terminal, inicie o servidor líder:

   ```
   python servidor_lider.py
   ```

3. Em outro terminal, inicie um ou mais servidores secundários:

   ```
   python servidor_secundario.py
   ```

4. Em outro terminal, execute um cliente:

   ```
   python cliente.py
   ```

5. No cliente, utilize os seguintes comandos:

   - `add;Comprar pão`
   - `remove;Comprar pão`
   - `edit;Comprar pão;Comprar café`
   - `list`
   - `exit` ou `quit` para encerrar

Etapa 3: Comunicação por Mensagens e Descoberta de Processos

Visão Geral da Implementação

Nesta etapa, a arquitetura do sistema foi refinada para adotar um modelo de comunicação baseado em mensagens estruturadas (no formato JSON), diferenciando comandos e eventos. Além disso, foi implementado um mecanismo de descoberta de serviços por meio de um **servidor de nomes**, responsável por registrar e localizar dinamicamente os componentes distribuídos no sistema.

Projeto de Arquitetura de Mensagem

Tipos de mensagem definidos:

- **comando**: mensagens enviadas pelos clientes ao servidor líder solicitando ações (ex: adicionar tarefa).
- **evento**: mensagens enviadas pelo líder para os servidores secundários, notificando mudanças no estado.
- **registro**: mensagens usadas para que um componente se registre no servidor de nomes.
- **consulta**: mensagens usadas para consultar o endereço (IP/porta) de um serviço.
- **resposta/erro**: mensagens enviadas como retorno estruturado para qualquer requisição.

Formato geral das mensagens (JSON):

```
{
  "tipo": "comando",
  "origem": "cliente",
  "comando": "add",
  "dados": {
    "tarefa": "Comprar pão"
  }
}
```

Fluxos de comunicação principais:

1. **Cliente → Servidor de Nomes**: envia uma consulta para descobrir o IP/porta do líder.
2. **Cliente → Servidor Líder**: envia comando estruturado para modificar ou consultar a lista.
3. **Servidor Líder → Réplicas**: envia eventos com as atualizações de estado.

Comunicação Assíncrona

A comunicação entre o servidor líder e os servidores secundários foi adaptada para o modelo assíncrono: o líder envia eventos imediatamente após processar uma modificação, sem aguardar resposta das réplicas. Este modelo promove desacoplamento entre os componentes e prepara o sistema para suportar escalabilidade e tolerância a falhas.

Mecanismo de Nomenclatura de Processos

Foi criado um novo componente, chamado **servidor de nomes**, responsável por armazenar registros de serviços com seus respectivos endereços.

Funcionamento:

- Ao iniciar, o **servidor líder** se registra no servidor de nomes como `"servidor_lider"`.
- Os **clientes** e **réplicas** realizam uma consulta ao servidor de nomes para obter o IP e a porta atual do líder.

Exemplo de mensagem de registro:

```
{
  "tipo": "registro",
  "nome": "servidor_lider",
  "ip": "127.0.0.1",
  "porta": 5000
}
```

Exemplo de mensagem de consulta:

```
{
  "tipo": "consulta",
  "nome": "servidor_lider"
}
```

Instruções de Execução Atualizadas

1. Inicie o servidor de nomes:

   ```
   python servidor_nomes.py
   ```

2. Em outro terminal, inicie o servidor líder (ele irá se registrar automaticamente):

   ```
   python servidor_lider.py
   ```

3. Em um ou mais terminais, inicie os servidores secundários:

   ```
   python servidor_secundario.py
   ```

4. Por fim, execute o cliente:

   ```
   python cliente.py
   ```

5. Utilize os comandos no cliente como anteriormente, por exemplo:

   - `add;Comprar pão`
   - `remove;Comprar pão`
   - `edit;Comprar pão;Comprar leite`
   - `list`

Etapa 4: Relógios Lógicos e Exclusão Mútua Distribuída

Visão Geral da Implementação

Nesta etapa, o sistema foi estendido para incluir **relógios lógicos de Lamport** e **relógios vetoriais**, que permitem estabelecer uma ordem parcial e total entre os eventos distribuídos. Esses mecanismos foram integrados em todas as mensagens trocadas entre clientes, líder e réplicas. Além disso, foi implementado um controle de **exclusão mútua distribuída** para garantir consistência nas operações de escrita (add, remove, edit) sobre a lista de tarefas.

Relógio de Lamport

Cada processo (cliente, líder, réplica) mantém um contador lógico:

- Antes de cada envio de mensagem, o contador Lamport é incrementado.
- Ao receber uma mensagem, o processo atualiza seu contador para o valor máximo entre o local e o recebido, acrescido de 1.

Este valor é incluído no campo `lamport` das mensagens JSON. Assim, é possível estabelecer uma ordem global dos eventos, ainda que não exista um relógio físico sincronizado.

Relógio Vetorial

Cada processo mantém também um vetor de tempos lógicos:

- O vetor possui uma entrada para cada processo participante conhecido.
- Em cada evento local ou envio de mensagem, o processo incrementa sua própria posição.
- No recebimento, os vetores são mesclados (máximo por posição) e incrementa-se a posição local.

Este valor é incluído no campo `vector` das mensagens JSON. Os relógios vetoriais permitem distinguir quando dois eventos são **concorrentes** (não há relação de causalidade) ou quando existe causalidade entre eles.

Exclusão Mútua Distribuída

Para garantir consistência na manipulação da lista de tarefas, foi implementado um mecanismo centralizado no líder:

- Cada operação de escrita (add, remove, edit) enviada por um cliente é registrada em uma fila de requisições do líder.
- A fila é ordenada pelo carimbo de tempo de Lamport, garantindo justiça e ordem consistente.
- O líder processa uma requisição por vez, enviando a resposta ao cliente e replicando o evento para os secundários.
- Operações de leitura (`list`) não passam pela fila e podem ser respondidas imediatamente.

Esse mecanismo garante que nunca duas operações de escrita concorrentes sejam aplicadas simultaneamente, preservando a integridade da lista.

Exemplo de Mensagem com Metadados de Relógio

```
{
  "tipo": "comando",
  "origem": "cliente",
  "id": "cli-8b665dbc",
  "comando": "add",
  "dados": {
    "tarefa": "Trabalho SD"
  },
  "lamport": 3,
  "vector": {"cli-8b665dbc": 1, "lider": 2}
}
```

Evidências de Funcionamento

Nos testes realizados, foi possível observar:

- Os valores de Lamport (`L=...`) aumentando a cada evento.
- Vetores de tempo crescendo e revelando relações de causalidade e concorrência.
- Em cenários com múltiplos clientes, o líder manteve a ordem justa das operações, aplicando as alterações uma por vez.

Instruções de Execução Atualizadas

As instruções de execução permanecem as mesmas da Etapa 3. Contudo, agora cada cliente e réplica exibem, junto às respostas e eventos, os valores dos relógios lógicos, permitindo verificar a evolução temporal distribuída:

```
[RESPOSTA] Tarefa adicionada: Trabalho SD
[LAMPORT] 4
[VETOR] {'cli-8b665dbc': 2, 'lider': 2}
```