Especificação do Projeto Final
Servidor de Espaço de Tuplas (Linda)

Disciplina de Programação Concorrente

1. Objetivo Geral

O projeto consiste em implementar um **servidor concorrente** que disponibiliza um **espaço de tuplas** (tuple space) inspirado no modelo Linda, oferecendo as operações:

- **WR**: escrita de tupla;
- **RD**: leitura não destrutiva, bloqueante;
- **IN**: leitura destrutiva, bloqueante;
- **EX**: execução de serviço sobre uma tupla de entrada, produzindo uma tupla de saída.

O servidor deve ser implementado em quatro linguagens, em versões independentes: **C++**, **Rust**, **Go** e **Elixir**. Cada versão deve obedecer à mesma semântica e ao mesmo protocolo.

2. Modelo de Tupla

Cada tupla possui o formato:

$$(chave : string,\ valor : string)$$

- A chave é uma string arbitrária.
- O valor é uma string arbitrária.
- O espaço pode conter várias tuplas com a mesma chave.
- Não há interpretação semântica do valor, exceto na operação `EX`.

3. Semântica das Operações

3.1 WR(k, v)

- Insere a tupla (k, v) no espaço.
- Não bloqueia e nunca falha.
- Retorna `OK`.

3.2 RD(k)

- Se existe alguma tupla com chave k, retorna seu valor sem removê-la.
- Caso contrário, a chamada deve **bloquear** até existir tal tupla.
- A política de seleção é FIFO por chave: retorna a tupla mais antiga.

3.3 IN(k)

- Igual a RD(k), mas remove a tupla retornada.
- Também é bloqueante.
- A política de remoção é FIFO.

3.4 EX(k_in, k_out, svc_id)

a) A chamada bloqueia até existir uma tupla (k_in, v).

b) Remove essa tupla, como em IN.

c) Verifica se há serviço associado a `svc_id`.

d) Caso o serviço exista, aplica a função:

$$v_{out} = servico(v)$$

e) Insere (k_out, v_{out}) no espaço.

f) Caso o serviço não exista, retorna `NO-SERVICE` e nada é inserido.

4. Serviços Registrados

O servidor mantém uma tabela estática:

$$svc\_id \longrightarrow funcao(string) \longrightarrow string$$

- Deve haver ao menos três serviços implementados.
- Exemplos possíveis:
  - `1`: converter a string para maiúsculas;
  - `2`: inverter a string;
  - `3`: retornar o tamanho da string como valor textual.
- Serviços inexistentes são ignorados, retornando `NO-SERVICE`.

5. Concorrência e Sincronização

Requisitos obrigatórios:

- O servidor deve aceitar **múltiplas operações simultâneas**.
- Estruturas internas devem ser protegidas adequadamente:
  - C++: mutexes e condition variables;
  - Rust: canais ou `Mutex+Condvar` (sem uso de `unsafe`);
  - Go: goroutines e canais ou mutexes padrão;
  - Elixir: processos e mailboxes (GenServer recomendado).
- Operações bloqueantes não podem impedir o funcionamento global do servidor.
- É proibido busy-waiting.

6. Protocolo de Comunicação

O servidor deve operar sobre TCP, recebendo comandos textuais no formato:

```
WR chave valor
RD chave
IN chave
EX chave_entrada chave_saida svc_id
```

Respostas esperadas:

- `WR`: retorna `OK`
- `RD` e `IN`: retorna `OK valor`
- `EX` com serviço existente: `OK`
- `EX` com serviço inexistente: `NO-SERVICE`
- Comando inválido: `ERROR`

7. Execução

Cada implementação deve:

- Abrir uma porta TCP definida em arquivo de configuração ou constante;
- Atender múltiplos clientes simultaneamente;
- Registrar apenas mensagens de erro interno no terminal;
- Ser entregue com instruções claras de compilação e execução.

8. Restrições

- É proibido utilizar bancos de dados ou bibliotecas externas de armazenamento.
- Deve-se usar apenas bibliotecas padrão das linguagens.
- Em Rust é proibido o uso de `unsafe`.

9. Critérios de Avaliação

Critério	**Peso**
Semântica correta das operações	3
Implementação correta da concorrência	3
Implementação correta do protocolo TCP	2
Qualidade dos serviços implementados	1
Clareza, organização e estilo do código	1

10. Entrega

O grupo deve entregar:

- Código-fonte das quatro versões: C++, Rust, Go e Elixir;
- README único contendo:
  - instruções de compilação e execução;
  - portas utilizadas;
  - descrição dos serviços;
  - exemplos mínimos de interação via TCP.

Entregar o link para o GitHub.

11. Programa Teste

Utilize esse programa para testar, como cliente, a sua solução. Importante: ainda não fiz o servidor... talvez esse programa de teste mude... Vou mantendo avisados.

Compile assim:

```
g++ -std=c++17 tester_linda.cpp -o tester_linda
```

Execute assim:

```
./tester_linda 127.0.0.1 54321
```

Atenção, opte por uma porta entre 49152 e 65535, em geral não dá ruim.

```cpp
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>

#include <cstring>
#include <iostream>
#include <sstream>
#include <string>

// Envia uma linha (cmd + '\n') e lê uma linha de resposta
std::string send_command(int sock, const std::string &cmd) {
    std::string to_send = cmd + "\n";
    ssize_t total_sent = 0;
    while (total_sent < static_cast<ssize_t>(to_send.size())) {
        ssize_t sent = ::send(sock, to_send.data() + total_sent,
                              to_send.size() - total_sent, 0);
        if (sent <= 0) {
            throw std::runtime_error("Erro ao enviar comando ao servidor");
        }
        total_sent += sent;
    }

    std::string response;
    char ch;
    while (true) {
        ssize_t rec = ::recv(sock, &ch, 1, 0);
        if (rec <= 0) {
            throw std::runtime_error("Conexao encerrada pelo servidor");
        }
        if (ch == '\n') {
            break;
        }
        if (ch != '\r') {
            response.push_back(ch);
        }
    }
    return response;
}

void expect_prefix(const std::string &resp, const std::string &prefix,
                   const std::string &context) {
    if (resp.rfind(prefix, 0) != 0) {
        std::cerr << "[FALHA] " << context
                  << " resposta inesperada: \"" << resp << "\"\n";
    } else {
        std::cout << "[OK] " << context
                  << " resposta: \"" << resp << "\"\n";
    }
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Uso: " << argv[0] << " <host> <porta>\n";
        std::cerr << "Exemplo: " << argv[0] << " 127.0.0.1 12345\n";
        return 1;
    }

    std::string host = argv[1];
    std::string port = argv[2];

    // Cria socket e conecta
    addrinfo hints{};
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    addrinfo *result;
    int ret = ::getaddrinfo(host.c_str(), port.c_str(), &hints, &result);
    if (ret != 0) {
        std::cerr << "getaddrinfo: " << gai_strerror(ret) << "\n";
        return 1;
    }
```

```
int sock = -1;
for (addrinfo *rp = result; rp != nullptr; rp = rp->ai_next) {
    sock = ::socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sock == -1)
        continue;
    if (::connect(sock, rp->ai_addr, rp->ai_addrlen) == 0) {
        break;  // conectou
    }
    ::close(sock);
    sock = -1;
}
freeaddrinfo(result);

if (sock == -1) {
    std::cerr << "Nao foi possivel conectar ao servidor\n";
    return 1;
}

try {
    std::cout << "Conectado a " << host << ":" << port << "\n";

    // 1) Teste básico de WR e RD
    {
        std::string cmd = "WR teste1 valor1";
        std::string resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "WR teste1");

        cmd = "RD teste1";
        resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "RD teste1");
        std::cout << "  (RD teste1 retornou: \"" << resp << "\")\n";
    }

    // 2) Teste de IN removendo a tupla
    {
        std::string cmd = "IN teste1";
        std::string resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "IN teste1");
        std::cout << "  (IN teste1 retornou: \"" << resp << "\")\n";
    }

    // 3) Teste de EX com svc_id = 1
    {
        std::string cmd = "WR in1 abcdef";
        std::string resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "WR in1");

        cmd = "EX in1 out1 1";
        resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "EX 1");

        cmd = "RD out1";
        resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "RD out1 apos EX 1");
        std::cout << "  (RD out1 apos EX 1 retornou: \"" << resp << "\")\n";
    }

    // 4) Teste de EX com svc_id = 2
    {
        std::string cmd = "WR in2 ghijkl";
        std::string resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "WR in2");

        cmd = "EX in2 out2 2";
        resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "EX 2");

        cmd = "RD out2";
        resp = send_command(sock, cmd);
        expect_prefix(resp, "OK", "RD out2 apos EX 2");
        std::cout << "  (RD out2 apos EX 2 retornou: \"" << resp << "\")\n";
    }
```

```
        // 5) Teste de EX com servico inexistente
        {
            std::string cmd = "WR in3 xyz";
            std::string resp = send_command(sock, cmd);
            expect_prefix(resp, "OK", "WR in3");

            cmd = "EX in3 out3 99";  // supondo que 99 nao exista
            resp = send_command(sock, cmd);
            expect_prefix(resp, "NO-SERVICE", "EX 99");

            // opcional: tentar RD out3, deve bloquear se implementado corretamente
            // entao NAO fazemos RD out3 aqui para evitar travar o tester
        }

        std::cout << "Testes basicos concluidos.\n";
    } catch (const std::exception &e) {
        std::cerr << "Erro: " << e.what() << "\n";
        ::close(sock);
        return 1;
    }

    ::close(sock);
    return 0;
}
```