Universidade Estadual de Campinas
Instituto de Computação

Sistema de Gerenciamento de Filmes com Arquitetura Cliente-Servidor
Projeto 01 MC833

Thaís Steinmuller Farias
RA: 224707
Natalia Piva Vilela
RA: 204001

1 Introdução

O objetivo do projeto descrito no presente relatório é descrever e explicar a implementação de uma aplicação cliente-servidor desenvolvida em linguagem C para gerenciar um catálogo de filmes. Ele permite que usuários criem, consultem, editem e removam registros de filmes armazenados em um banco de dados SQLite. A comunicação entre cliente e servidor se dá via sockets TCP, e o sistema é projetado para suportar múltiplos acessos concorrentes via threads.

2 Rodando o Sistema

Nosso sistema é composto por dois programas: server e client, utilizando SQLite3 para persistência de dados. A compilação é automatizada via Makefile.

Pré-requisitos:

- Compilador gcc
- Biblioteca SQLite3

Para compilar o sistema, basta executar "make" e executar ./server e ./client em terminais ou máquinas distintas.

3 Arquitetura

O sistema segue uma arquitetura cliente-servidor, implementada conforme diagrama da Figure 1.

No código, os componentes principais estão divididos em:

- **Cliente**: Implementado em cliente.c, oferece uma interface no terminal bash quando está rodando para que o cliente possa interagir com o sistema conforme seu interesse. Responsável por traduzir comandos em requisições e exibir as respostas das requests ao usuário. Ele lista comandos disponíveis, coleta dados

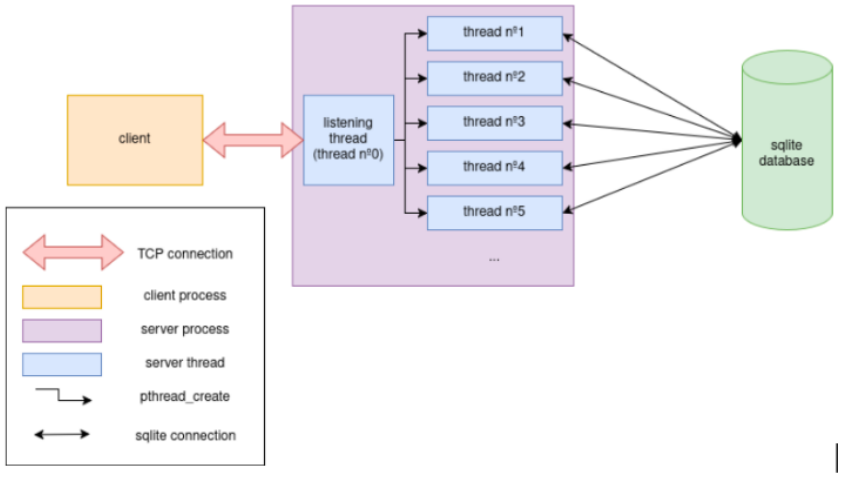

Figure 1: Representação gráfica do modelo de arquitetura implementado

.

via terminal, usa snprintf para montar o body com delimitadores e envia e recebe dados via **send()** e **recv()**, com verificação de bytes transmitidos.

- **Servidor**: Implementado em server.c, é responsável por receber e gerenciar conexões de clientes via sockets TCP, recebe e executa operações no banco de dados e envia respostas. Utiliza de threads para atender múltiplos clientes ao mesmo tempo, garantindo escalabilidade. Ele escuta na porta 7080, aceitando múltiplas conexões usando pthread; em seguida cada thread executa respond_to_request() e usa o SQLite em modo parallel com uma conexão por thread. Por fim, implementa execute_command() para tratar operação e responde utilizando os headers e dados formatados pelo string_t.

- **Banco de Dados**: Interface de um banco de dados de filmes implementada em database.c. Usamos a biblioteca SQLite para armazenamento persistente dos dados.

- **Protocolo de Comunicação**: Arquivos request.c e request.h estabelecem o formato das mensagens trocadas entre cliente e servidor, dando suporte à comunicação.

4 Estrutura de Armazenamento

Como mencionado na seção anterior, os dados de filmes são armazenados no banco de dados SQLite, escolhido por ser simples de usar e configurar e leve. A tabela responsável por armazenar os dados de filmes no código é criada em:

```
    CREATE TABLE IF NOT EXISTS films (
    title TEXT,
    genre TEXT,
    director TEXT,
    year INT
);
```

Sendo films a struct que define o modelo do filme guardado, como o código abaixo:

```
  struct film {
  int id;
  string_t title;
  string_t genre;
  string_t director;
  int year;
};
```

Como podemos ver, cada filme possui o atributo

- title: Título do filme
- genre: Gênero do filme, podendo ser mais de um.
- director: Diretor do filme
- year: Ano de lançamento do filme
- rowid: Identificador único que é gerado pelo SQLite de maneira automática.

É importante frisar que a persistência de dados é garantida pelo gerenciamento do próprio SQLite, que usa prepared statements para intergir com o banco, promovendo segurança contra SQL injections.

5 Operações

As seguintes operações foram implementadas, conforme pedido no projeto:

- CREATE_FILM: Adiciona novo filme ao banco de dados. Recebe como parâmetros o título, gênero, diretor e ano e retorna o ID do filme criado.
- REMOVE_FILM: Remove filme já existente do banco de dados. Recebe como parâmetros o ID do filme e retorna confirmação ou código de erro, referentes à operação.
- ADD_GENRE: Adiciona novo gênero a um filme existente. Recebe como parâmetros o ID do filme e gênero e retorna uma confirmação ou um código de erro.
- LIST_TITLES: Lista todos os IDs e títulos dos filmes. Não possui parâmetros e retorna lista de IDs e títulos.
- LIST_FILMS:Lista todas as informações referentes à struct film presentes. Não tem parâmetros e retorna a lista completa de informações dos filmes.
- GET_FILM: Recupera informações detalhadas de um filme específico, tomando como parâmetro o ID do filme. Retorna informações completas sobre esse filme.
- LIST_BY_GENRE: Lista filmes que contém um determinado gênero. O gênero a pesquisar é o parâmetro, retornando a lista de filmes que o correspondem.

6 Bibliotecas Utilizadas

As bibliotecas usadas para controle de comunicação de sockets e redes, conforme vistas em aula foram <sys/socket.h>, <netinet/in.h> e <arpa/inet.h>. Adicionalmente,<pthread.h> foi usada para processamento paralelo de múltiplas conexões e <sqlite3.h> para gerenciamento de banco de dados. O controle de erros é feito em when_macros.h e a manipulação de strings implementada em string.h.

7 Formatação de Request e Resposta

A comunicação segue o seguinte design:

- Request:
  - **Header**: Descrita em request_header_t que contêm o comando (command_e) e o tamanho do body.
  - **Body**: Contêm parâmetros necessários para a operação.
- Resposta:
  - **Header** Descrita em response_header_t: Contêm o código de resposta (response_code_e), contador e tamanho do body.
  - **Body**: Contêm os dados solicitados

Os dados no corpo das mensagens são estruturados usando separadores especiais:

- BODY_FIELD_SEPARATOR (1F): Separa campos de um registro
- BODY_RECORD_SEPARATOR (1E): Separa registros diferentes

8 Mecanismo de Threads

Para lidar com múltiplas conexões dos clientes, o servidor implementa um modelo de multithreading baseado em threads POSIX (pthreads), onde cada conexão de cliente é tratada por uma thread dedicada. No arquivo server.c, podemos observar a implementação deste mecanismo:

```
while (1) {
    socklen_t clilen = sizeof(cliaddr);
    int res_fd = accept(sock_fd, (struct sockaddr *)&cliaddr, &clilen);
    if (-1 == res_fd) {
      perror("accept");
      goto error;
    }
    fprintf(stderr, "INFO: A new client connected\n");
    // Create a new thread and pass it the socket file descriptor
    pthread_create(&thread, NULL, respond_to_request, (void *)(uintptr_t)res_fd);
    pthread_detach(thread);
}
```

O servidor cria uma nova thread para cada conexão aceita e imediatamente a "destaca" (pthread_detach), permitindo que ela execute de forma independente e seja automaticamente liberada quando sua função terminar, sem necessidade de um pthread_join explícito.

Escolhemos threads para paralelização por terem menor custo de criação e troca de contexto. Isso permite ao servidor atender múltiplas conexões simultaneamente, melhorando a escalabilidade do sistema.

9 Error Handling

Implementamos os seguintes mecanismos de tratamento de erros:

- Macros personalizadas como when_false_ret e when_null_jmp para verificação de condições
- Códigos de erro padronizados (NO_ERROR, INTERNAL_ERROR, ERROR_NOT_FOUND
- Registro de erros no stderr para facilitar a depuração

10 Casos de Uso

A seguir encontram-se detalhados os pipelines para dois dos casos de usos pedidos no projeto, para melhor compreensão do funcionamento do sistema e de sua integração:

10.1 Caso 1: Cadastrando um Novo Filme

Cenário: Um usuário deseja adicionar o filme "O Poderoso Chefão" ao banco de dados.

Pipeline:

1. O usuário inicia o cliente e conecta-se ao servidor: ./client 127.0.0.1:7080
2. O cliente exibe um menu com opções disponíveis
3. O usuário seleciona a opção 0 (CREATE_FILM)

4. O cliente solicita os dados do filme, como por exemplo:

   ```
   Título: O Poderoso Chefão
   Gênero: Crime,Drama
   Diretor: Francis Ford Coppola
   Ano: 1972
   ```

5. O cliente constrói uma requisição com o comando CREATE_FILM e um corpo contendo os dados formatados com separadores de campo.

6. O servidor recebe a requisição, executa o comando database_insert_film e devolve um ID único (ex: 123)

7. O cliente informa ao usuário que o filme foi cadastrado com sucesso.

Nível de Rede:

- O cliente envia um header request_header_t com command = CREATE_FILM e body_size = tamanho do corpo
- O corpo contém:

  ```
  "O Poderoso Chefão\x1FCrime,Drama\x1FFrancis Ford Coppola\x1F1972"
  ```

- O servidor responde com response_header_t contendo code = NO_ERROR e count = 1 (ID do filme)

10.2 Caso 2: Adicionando um Novo Gênero a um Filme

Cenário: Queremos adicionar o gênero "Clássico" ao filme "O Poderoso Chefão".
Pipeline:

1. O usuário lista os filmes disponíveis usando a opção 3 (LIST_TITLES) para identificar o ID do filme

   ```
   | 1 | O Poderoso Chefão |
   | 2 | Pulp Fiction |
   ```

2. O usuário seleciona a opção 2 (ADD_GENRE)

3. O cliente faz solicitação via bash:

```
ID do filme: 1
Gêneros a adicionar: Clássico
```

4. O servidor executa a seguinte sequência:
   - Inicia uma transação.
   - Obtém o valor atual do campo "gênero" (Crime,Drama).
   - Adiciona o novo gênero com vírgula (Crime,Drama,Clássico).
   - Atualiza o registro e finaliza a transação.
5. O cliente recebe confirmação de sucesso

   A função string_join nesse caso gerencia a concatenação dos valores com delimitadores apropriados

11 Conclusão

O sistema desenvolvido por nós demonstra uma implementação eficiente e funcional de uma aplicação cliente-servidor para gerenciamento de filmes. A arquitetura adotada permite escalabilidade de tamanho através do uso de **threads** para atender múltiplos clientes. O protocolo de comunicação é bem definido e suporta várias operações, possuindo tratamento adequado de erros.

Consideramos como pontos fortes da implementação o uso de threads para concorrência, a estruturação do protocolo de comunicação TCP, as operações transacionais no banco de dados (especialmente em ADD_GENRE) e o tratamento consistente de erros. Pontos para melhoria no sistema a serem tratados no em um cenário de escalabilidade seriam, por exemplo, a compressão dos dados transmitidos e implementação de cache de consultas frequentes.

O sistema demonstra claramente os conceitos fundamentais de programação de redes, incluindo comunicação socket, paralelismo, persistência de dados e desenvolvimento de protocolos.