Danilo Dainez da Rocha 9875458
Lucas Leal Caparelli 9844957

Relatório DSID

São Paulo

2018

Danilo Dainez da Rocha 9875458
Lucas Leal Caparelli 9844957

Relatório DSID

Relatório referente ao trabalho da disciplina - Desenvolvimento de Sistemas de Informação Distribuídos

Universidade de São Paulo – USP

Escola de Artes, Ciências e Humanidades

Bacharelado em Sistemas de Informação

São Paulo

2018

RESUMO

O presente trabalho tem como objetivo discorrer sobre o funcionamento e a implementação de um sistema distribuído *multithreaded* de cliente-servidor. Escolheu-se como aplicação um servidor capaz de gerenciar em memória informações quanto a contas bancárias. Estas contêm: nome completo do titular, endereço do titular e saldo do titular. O cliente, por sua vez, é capaz de criar, atualizar, obter e deletar contas bancárias.

Palavras-chaves: Cliente-Servidor. *Multithreaded*. Sistema Distribuído.

SUMÁRIO

INTRODUÇÃO **4**

1 A APLICAÇÃO **5**

2 DETALHES DE IMPLEMENTAÇÃO **6**
2.1 Server **6**
2.1.1 Service 7
2.1.2 Model 8
2.2 Client **8**

3 REGISTRO DE DECISÕES DE ARQUITETURA **9**
3.1 Decisões gerais **9**
3.2 Threading **9**
3.3 Estruturas globais e de sincronização **10**
3.4 Cliente **11**
3.5 Uso e testes **12**

4 CONCLUSÃO **13**

INTRODUÇÃO

O objetivo do projeto é a criação de um sistema distribuído de arquitetura cliente-servidor que faça uso de uma implementação de *threads* POSIX (IEEE Std 1003.1c, 1995).

Para tal escolheu-se usar C/C++ importando a biblioteca pthread.h, nativa da linguagem. Escolheu-se também trabalhar entre as camadas de rede de transporte e aplicação, fazendo uso de uma biblioteca para encapsulamento das operações oferecidas pela API de *sockets* nativa do C.

A aplicação busca gerenciar contas bancárias com os seguintes atributos:

- Nome completo do titular
- Endereço do titular
- Saldo do titular

Sendo o servidor capaz de receber requisições de:

- Criação de contas
- Atualização de contas
- Obtenção de contas
- Deleção de contas

1 A APLICAÇÃO

A estrutura do projeto é:

```
|--- client
|--- logs
|--- server
    |--- json
    |--- model
    |--- service
|--- socket
```

- Client: Contém a aplicação cliente que recebe o *input* do usuário, valida esse *input* e o envia ao servidor.
- Logs: Contém o arquivo de *logs* do servidor.
- Server: Pacote que contém o servidor em si.
- json: Biblioteca para serialização de desserialização de JSONs (*JavaScript Object Notation*).
- Model: Pacote que mantém o modelo de dados de conta e a estrutura de dados onde essas contas são armazenadas.
- Service: Pacote que encapsula a lógica da aplicação, contendo a implementação das operações de CRUD (*Create* - *Read* - *Update* - *Delete*) do servidor.
- Socket: Biblioteca utilizada tanto pelo servidor quando pelo cliente, fornece uma abstração em cima dos *sockets* nativos do SO para facilitar o uso desse recurso.

2 DETALHES DE IMPLEMENTAÇÃO

2.1 SERVER

Na raiz do pacote estão as classes *ServerEndpoint* e *Logger*.

- *ServerEndPoint*: O *Endpoint* da aplicação, resposável por estabelecer a conexão com os clientes através de *sockets* e por receber as requisições. Essa classe também contém a logica de criação de *threads*, mantendo a *thread pool* e alocando uma *thread* para cada requisição. Possui também variáveis globais contendo a estrutura de dados utilizada para armazenar as contas bancárias, a fila global de associada às requisições e as variáveis globais de sincronização de *threads*.
- Logger: Classe utilitária, escreve os *logs* em um *buffer* em memória e é capaz de gravar esse *buffer* em um arquivo de texto.

Funções presentes em ServerEndpoint:

- main: função principal do servidor. Responsável por criar um socket TCP/IP associado à porta 8080, inicializar as variáveis de sincronização, criar o *pool* de *threads* e então receber conexões até o fim de sua execução, gerada pela interrupção do usuário. Sempre que recebe uma requisição, insere o socket gerado para envio da resposta numa fila FIFO (*first-in-first-out*) e então sinaliza uma das *threads* bloqueadas que há trabalho a ser feito. Caso não haja nenhuma disponível (*i.e.*, que já não esteja trabalhando em outra requisição) então é gerada uma nova *thread* efêmera que será responsável por tratar essa única requisição e então será finalizada.
- work_on_request: rotina a ser executada pelas *threads* pertencentes ao *pool*. É iniciada bloqueando a *thread* até que seja sinalizada pelo main que há trabalho a ser feito. A *thread* que é desbloqueada então consome um elemento da fila de *sockets* associada à requisição. Então trata a requisição através da função process_request, descrita posteriormente nesta seção. Feito isso envia a resposta ao cliente através do *socket* adquirido e então é bloqueada novamente até que seja sinalizada uma vez mais. Essa função recebe como argumento um ID de 0 até o número de *threads* estabelecido pela constante NUM_THREADS. Este ID é utilizado para captura de *logs*.
- create_ephemeral_thread: rotina executada por *threads* não pertencentes ao *pool*. É chamada apenas quando todas *threads* do *pool* estão ocupadas atendendendo a outras requisições. Ela também consome um *socket* da fila FIFO, o utiliza para receber a

requisição, faz uso da função process_request para lidar com ela, envia a resposta através do *socket* e então encerra-se.

- process_request: recebe uma string contendo a requisição e então identifica a operação a ser realizada. Feito isso desserializa a requisição e chama as funções da classe AccountService para operar os dados. É responsável por retornar a resposta provida por essas funções.

Funções presentes em Logger:

- get_timestamp: responsável por adquirir a data e hora para uso nas entradas do log.
- write_to_log: recebe uma string que é a entrada a ser inserida no log. Escreve no *buffer* a entrada precedida pela hora e data atual, adquirida através da função descrita acima.
- flush_logs_to_file: realiza a escrita dos dados contidos no *buffer* no arquivo de *logs* e então limpa o *buffer*.

2.1.1 SERVICE

No pacote *Service*, temos a classe *AccountService* que fornece a implementação das operações de *CRUD*, essa classe contém as funções:

- create_account: Recebe por parâmetro a requisição, cria um novo objeto *Account* com as informações da requisição e adiciona esse objeto ao vetor de contas.
- update_account: Recebe por parâmetro o id da conta a ser atualizada e as informações a serem atualizadas na conta. Primeiro o método busca a conta pelo id e verifica se ela existe, se ele existir utiliza a referência recuperada na busca para atualizar as informações requisitadas pelo usuário.
- delete_account: Recebe por parâmetro o id da conta a ser deletada, busca essa conta pelo o id e se a conta for encontrada ela é removida do vetor de contas.
- find_account: Recebe por parâmetro o id da conta a ser encontrada, busca a conta pelo id e se a conta for encontrada ela é suas informações são enviadas como resposta.
- get_account_iterator: Função auxiliar para buscar contas, recebe o id de conta desejado por parâmetro e retornará um iterador já na posição onde a conta com esse id se encontra no vetor. Caso a conta nao seja encontrada o iterador estará após o fim do vetor de contas.
- does_account_exists: Função booleana auxiliar para verificar se uma conta existe. Se o iterador recebido por parâmetro não estiver após o fim do vetor, a conta não existe, caso o contrário a conta existe.

- floatToString: Função para converter float para string, utilizada ao montar as respostas.

2.1.2 MODEL

Contém o modelo de dados, que consiste do objeto *Account* que contém as propriedades:

- *id*
- *name*
- *address*
- *balance*

Na classe *AccountModel*, estão contidos, além das propriedades em si, os *getters* e o *setters* para essas propriedades. Sendo o objeto *Account* a representação de uma conta de usuário dentro do sistema.

2.2 CLIENT

O pacote *client* contém apenas um único objeto, contendo uma função main, uma função is_digits e uma função getRequest. A descrição destas são:

- main: responsável por gerar o *socket* que será utilizado na comunicação com o servidor, chamar a função get_request, escrever a requisição no *socket*, receber a resposta e a imprimí-la na saída padrão.
- is_digits: função booleana que recebe um ponteiro para uma string. Responsável por verificar se a string recebida é composta apenas por dígitos numéricos. Utilizada para validação do saldo informado pelo usuário.
- getRequest: função que retorna uma string e não recebe nada como parâmetro. Responsável por coletar da entrada padrão as informações necessárias para criação da requisição, validar a entrada e criar um JSON que será enviado ao servidor como requisição.

3 REGISTRO DE DECISÕES DE ARQUITETURA

Este capítulo aborda decisões tomadas ao longo da implementação do projeto no intuito de apresentar a racionalização estabelecida ao tomá-las, tendo em vista o escopo da disciplina, os objetivos de implementação, critérios de avaliação, bem como limitação de tempo.

3.1 DECISÕES GERAIS

- Língua do código: escolheu-se redigir o código e seus comentários utilizando a língua inglesa, pois ambos autores têm intenção de fazer uso do código produzido em seu portfólio público. Com a finalidade de alcançar um maior público dentro da comunidade FOSS (*Free and Open Source Software*) escolheu-se o inglês.
- Linguagem de programação do código: escolheu-se fazer uso de C/C++ devido ao fato de ser a implementação mais bem difundida e bem documentada de *threads* POSIX além do fato de serem linguagens familiares aos autores.
- Bibliotecas: fez-se uso de bibliotecas para manipulação de *sockets* e para manipulação de JSON. Essa decisão foi tomada pelo fato de que a definição do protocolo de comunicação na camada de aplicação não tangencia o conteúdo da disciplina, de forma que fazer uso de bibliotecas nativas de baixo nível ou uma implementação do zero trariam faria apenas com que mais tempo fosse gasto desenvolvendo atividades que não contribuem para o aprendizado do tema central abordado pela disciplina.
- *Logging*: fez-se o *log* da aplicação nos seguintes momentos: durante a criação das *threads* permanentes, durante o recebimento de requisições em cada uma das *threads* (efêmeras ou não) e ao se enviar a resposta ao cliente. Prefixou-se cada entrada do *log* com data e horário a fim de facilitar a análise do paralelismo presente no programa.
- *Buffer* do *log*: criou-se um buffer para os logs a fim de reduzir-se o uso de I/O no programa, visto a perda significativa de desempenho numa aplicação de alta intensidade de IO.

3.2 THREADING

- Quantidade de *threads*: escolheu-se definir a quantidade de *threads* através de uma constante no arquivo server/ServerEndpoint.cpp NUM_THREADS. A ideia é permitir

que isso seja facilmente alterado e possivelmente definido através de um arquivo de configuração no futuro. Atribui-se a essa constante o valor 10 inicialmente, visto que não espera-se que o projeto seja utilizado para fins reais, mas que sirva apenas para demonstração do uso de pthreads. Um número baixo facilitaria a análise do comportamento do programa e suas *threads*.

- *Pool* de *threads*: optou-se pela criação de um *pool* de *threads*. A intenção dessa decisão é que elimine-se o *overhead* de criação de uma *thread* para cada requisição, aperfeiçoando então o desempenho do programa. Observa-se que o ganho não é tão significativo quando comparado ao ganho de se utilizar paralelismo através de *threads* ao invés de processos, porém considerou-se significativo o suficiente tendo em vista uma aplicação em produção recebendo milhares de requisições simultaneamente.

- *Threads* efêmeras: quando todas *threads* do *pool* estão ocupadas é gerada uma nova *thread* efêmera para tratar a nova requisição. Esta *thread* é encerrada assim que finaliza o processamento desta requisição. Optou-se por essa estratégia a fim de evitar que as requisições ficassem em uma fila aguardando tratamento. No dilema entre desempenho e experiência de usuário escolheu-se priorizar a experiência de usuário. Contudo, visto que o número de *threads* no *pool* é customizável torna-se possível o ajuste desse valor até que encontre-se um equilíbrio entre consumo de recursos (quanto maior o valor, mais *threads* em memória com seus próprio segmentos de dados e pilha de execução) e desempenho (quanto maior o valor, menor o número de *threads* efêmeras a serem criadas, reduzindo o *overhead*).

3.3 ESTRUTURAS GLOBAIS E DE SINCRONIZAÇÃO

- Vetor de contas: estrutura de dados globais presente no endereçamento de memória compartilhado do processo (*i.e.*, acessível a todas *threads*) que armazena os objetos das contas bancárias. Trata-se de um vetor de alocação dinâmica.

- Fila de *sockets*: uma das estruturas de dados globais, é uma fila de política FIFO (*first-in-first-out*) de *sockets*. Ela representa na verdade uma fila de requisições a serem consumidas pelas *threads* na qual o *socket* resultante da função accept() invocada no main é armazenado. Escolheu-se este método a fim de promover o desacoplamento referencial e temporal entre a *thread* principal e as trabalhadoras. Permite com que se guarde o dispositivo de comunicação com cada cliente numa estrutura sem ter conhecimento de qual trabalhadora irá processar a requisição e de forma que não importa se há uma trabalhadora pronta no momento ou não.

- Contador de *threads* disponíveis: variável global responsável por manter um registro de quantas *threads* do *pool* estão atualmente disponíveis. O desacoplamento referencial

promovido pela fila de *sockets* em conjunto com a maneira com a qual a especificação POSIX trata sinalização de *threads* bloqueadas a espera de uma *condition variable* permite a ausência de uma estrutura para gerenciamento dessas trabalhadoras. Com isso poupa-se na complexidade espacial do programa diretamente proporcional ao número de trabalhadoras definido, ocupando 32 bits em memória de maneira invariante.

- global_id: variável global chave única de identificação de cada conta presente em memória. É sempre apenas incrementada, de forma que mesmo que uma conta seja deletada nenhuma outra contada criada subsequentemente receberá o mesmo ID. Esta decisão foi tomada com a finalidade de manter a consistência lógica entre todas contas de forma que um id está associado, a todos momentos, apenas a uma única conta ou nenhuma (ainda não foi utilizado), não sendo possível que um id identifique uma conta A em determinado momento e passe a identificar uma conta B em outro momento.

- variáveis de sincronização: a cada uma dessas variáveis globais descritas acima está associada ao menos um dispositivo de comunicação entre as *threads* com o intuito de sincronizar as trabalhadoras. Todos possuem ao menos um mutex para agir como um semáforo em regiões críticas do código, impedindo inconsistências decorrentes de condições de corrida do tipo *read-write* e *write-write*. Além disso há também uma condition variable associada à fila de *sockets* que tem como trabalho indicar que há trabalho a ser feito. São estas variáveis de sincronização:

  - pthread_mutex_t account_mutex;
  - pthread_cond_t work_cond;
  - pthread_mutex_t sock_queue_mutex;
  - pthread_mutex_t id_mutex;
  - pthread_mutex_t thread_count_mutex;

3.4 CLIENTE

- Validação de requisições: optou-se por validar as requisições sempre que se pôde ainda no cliente, de forma a incumbir o servidor com o menor número de tarefas possíveis, visto que este tem como objetivo lidar com diversas requisições simultâneas de diversos clientes. Com isso, todas checagens de formatação e uso são feitas no cliente, restando apenas ao servidor realizar o processamento da requisição.

- O cliente realiza apenas uma requisição por execução. Tomou-se essa decisão tendo em vista manter a simplicidade do uso da API.

3.5 USO E TESTES

Informações e instruções quanto ao uso do programa, bem como testes a serem realizados estão presentes nos arquivos README.pdf e README.md, ambos disponibilizados em formatos diferentes para conveniência, mas apresentando mesmo conteúdo.

A suíte de testes de *stress* foi implementada em Kotlin, linguagem de programação multi-paradigma que é compilada em bytecode Java para posteriormente ser interpretado por uma JVM (Java Virtual Machine). A suíte realiza 1000 requisições com um intervalo de 10 ms entre cada uma. Este intervalo tem como propósito evitar problemas de escrita no *socket* causados pela limitação de recursos das máquinas nas quais o programa será executado. De qualquer maneira a requisição não é tratada em menos de 10 ms, de forma que o servidor trata mais de uma requisição ao mesmo tempo.

Para demais testes não relacionados à carga recomenda-se o uso do cliente em C/C++.

4 CONCLUSÃO

A implementação de um sistema distribuído não é trivial e com esse projeto buscou-se demonstrar que mesmo com uma logica de negócios extremamente simples existem diversas complicações geradas ao utilizar-se de paralelismo. Vale salientar, entretanto, que toda a complexidade adicional é decorrente apenas do uso de *threads* e que a situação seria bastante diferente para o caso de diversos nós com diversas *threads* em comunicação constante para manter a coesão do sistema. Neste ambiente a complexidade seria muito maior do que a apresentada aqui.

Acredita-se que a aplicação elaborada possui diversos pontos a melhorar, como por exemplo o tratamento de sinais transmitidos pelo S.O., adição de granularidade quanto à verbosidade dos logs (*e.g.*, modo debug, verboso, erros, informacional), persistência não-volátil dos dados, entre outras funcionalidades que seriam importantes do ponto de vista de lógica de negócios e uso do software como um serviço real. Entretanto, dado o escopo e objetivo da disciplina, acredita-se que a exploração da biblioteca pthreads juntamente aos problemas de sincronização propostos e as otimizações realizadas com o fim de promover maior desempenho e reduzir a complexidade atendem aos critérios de avaliação apresentados, assim como cumprem com os objetivos propostos no enunciado do trabalho.