# Trabalho de Projeto

3ª Fase

## 1 Descrição Geral

A componente teórico-prática da disciplina de sistemas distribuídos consiste no desenvolvimento de quatro projetos, utilizando a linguagem de programação C [4], sendo que a realização de cada um deles é necessária para a realização do projeto seguinte. Por essa razão, **é muito importante que consigam cumprir os objetivos de cada projeto, de forma a não hipotecar os projetos seguintes**.

O objetivo geral do projeto será concretizar um serviço de armazenamento baseado em pares chave-valor, nos moldes da interface *java.util.Map* da API Java e similar ao utilizado pela *Amazon DynamoDB* para dar suporte a muitos serviços Web [1]. Neste sentido, as estruturas de dados utilizadas para armazenar esta informação são uma **lista encadeada simples** [2] e uma ***hash table*** [3], dada a sua elevada eficiência ao nível da pesquisa.

Na 1ª fase do projeto foram definidas as estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na *hash table*. Também foi construído um módulo para a serialização de dados, que teve como objetivo familiarizar os alunos com a necessidade de serializar dados para a comunicação em aplicações distribuídas.

A 2ª fase do projeto teve como objetivo implementar um sistema cliente-servidor simples, sendo o servidor responsável por manter uma *hash table* (usando os módulos construídos na 1ª fase do projeto) e sendo o cliente responsável por comunicar com o servidor para realizar operações nesta tabela. Foi também utilizado o ***Protocol Buffers*** da Google [5] para automatizar a serialização e deserialização dos dados, tendo por base um ficheiro `htmessages.proto` com a definição da mensagem a ser usada na comunicação, tanto para os pedidos como para as respostas.

Na 3ª fase do projeto iremos criar um sistema concorrente que aceita e processa pedidos de múltiplos clientes em simultâneo através do uso de múltiplas *threads*. Mais concretamente, vai ser preciso:

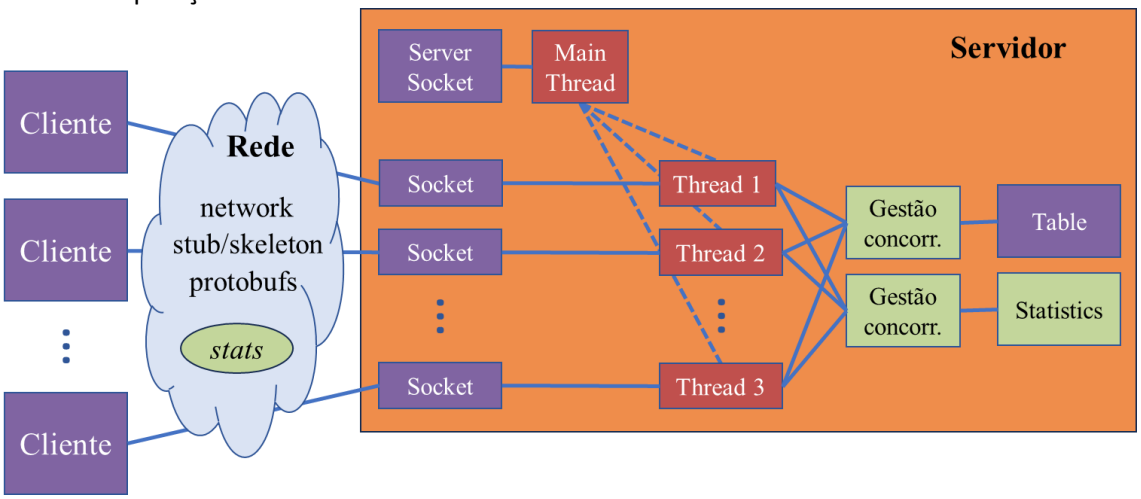
- Adaptar o servidor para este suportar múltiplos clientes ligados em simultâneo. Serão usadas *threads* para atender os clientes, seguindo um modelo *one-thread-per-client*;
- Implementar no servidor uma estrutura `statistics_t` para guardar as estatísticas permanentemente atualizadas sobre:
  - O número total de operações na tabela executadas no servidor;
  - O tempo total acumulado gasto na execução de operações na tabela;
  - O número de clientes atualmente ligados ao servidor.
- Implementar uma nova operação `stats`, que permita a um cliente obter as estatísticas atuais guardadas no servidor, alterando para tal o ficheiro `htmessages.proto` para suportar esta nova operação.
- Adaptar o servidor para ter as seguintes *threads*:
  - a *thread* principal do programa, que fica responsável por aceitar novas ligações de clientes e por criar uma *thread* para atender cada novo cliente;
  - *threads* secundárias, lançadas pela *thread* principal, que se vão manter ativas para atender os pedidos de um cliente até que este encerre a ligação.
- Realizar a gestão da concorrência no acesso das *threads* secundárias às estruturas partilhadas, nomeadamente à tabela e à estrutura `statistics_t` do servidor, usando mecanismos de gestão da concorrência, nomeadamente *mutexes* e/ou *condições*.

Como nos projetos anteriores, espera-se uma **grande fiabilidade** do servidor e do cliente, sendo para tal necessário tratar todas as condições de erro e garantir uma **correta gestão da memória**, evitando *memory leaks* ou acesso a zonas de memória inválidas. É importante notar que em sistemas cliente-servidor os servidores ficam em funcionamento permanente durante muito tempo (dias, meses ou até anos) e não é suposto que parem (*crash*) por causa de erros.

## 2 Descrição Específica

O objetivo específico da 3ª Fase do Projeto é desenvolver uma aplicação do tipo cliente-servidor capaz de suportar múltiplos clientes de forma síncrona. Para tal, para além de aproveitarem o código desenvolvido nas fases 1 e 2 do projeto, os alunos devem fazer uso das APIs para programação com *threads*, *mutexes* e/ou *condições*. A utilização de *threads* permitirá o atendimento de pedidos de clientes feitos de forma concorrente, e os *mutexes* e/ou as *condições* permitirão gerir a concorrência das *threads* no acesso às estruturas de dados partilhadas. Pretende-se também neste projeto que os alunos implementem uma nova operação no servidor, a operação `stats`, tendo para isso de alterar o

ficheiro `htmessages.proto`. A nova operação servirá para obter as estatísticas atualizadas sobre o funcionamento do servidor, sendo necessário acrescentar os devidos campos à mensagem anteriormente definida para a comunicação entre o cliente e o servidor.
A figura abaixo ilustra o novo modelo de comunicação que será usado na 3ª fase do projeto. Atenção que este modelo abstrai os detalhes de comunicação implementados na 2ª fase do projeto, colocando os módulos *network*, *stub* e *skeleton* na rede de interligação. A cor roxa representa o que já foi feito nas fases 1 e 2 do projeto, e as cores verde e vermelho representam o que é preciso fazer nesta 3ª fase do projeto. Também é necessário alterar o ficheiro `htmessages.proto` e fazer pequenas alterações no código do cliente e nos módulos *stub* e *skeleton* para implementar a nova operação `stats`.




## 2.1 Servidor *multi-threaded* com modelo *one-thread-per-client*

Para ser possível atender a vários clientes em paralelo, serão utilizadas *threads* do lado do servidor. O atendimento dos clientes deverá seguir um modelo *one-thread-per-client*, ou seja, de cada vez que um cliente se liga ao servidor, este deverá criar uma *thread* que ficará responsável por atender todos os pedidos deste cliente de forma exclusiva até este fechar a ligação, ou até ocorrer algum erro. A criação de *threads* é feita através da API de *threads* do UNIX (`pthreads`).
Assim, apresenta-se de seguida a ideia geral do novo código a ser desenvolvido na função `network_main_loop(int listening_socket, struct table_t *table)` do módulo `server_network.c` do servidor, bem como do código a ser executado pelas *threads* secundárias. Assume-se que a *socket* de escuta (`listening_socket`) recebida como argumento já foi preparada para a receção de pedidos de múltiplas ligações num determinado porto (determinado pelo método `listen`). Cabe aos alunos traduzir esta ideia geral para o seu próprio código C (ou inventar outro algoritmo).

```
/*
 *  Esboço do algoritmo a ser implementado na função network_main_loop
 */

while (1) {
      connsockfd = accept(listening_socket);
      cria nova thread secundária passando-lhe connsockfd;
}

/*
 *  Thread secundária de atendimento do cliente
*/

while ((message = network_receive(connsockfd)) !=NULL) { // cliente não fechou conexão
      invoke(message);                                   // Executa pedido contido em message
      network_send(connsockfd, message);                 // Envia resposta contida em message
}
close(connsockfd);                        // cliente fechou conexão ou ocorreu um erro,
termina a thread secundária;              // por isso fecha  ligação e termina a thread
```

De notar que o atendimento de cada pedido será feito de forma síncrona, tal como já acontecia na 2ª fase do projeto. Isto significa que quando um cliente faz um pedido, ficará boqueado à espera da respetiva resposta.

## 2.2 Estatísticas do Servidor

Nesta fase do projeto, o servidor terá de manter algumas informações estatísticas sobre o seu funcionamento. Estas informações poderão estar agrupadas numa estrutura a ser partilhada por todas as *threads*.
O tipo desta estrutura (`struct statistics_t`) poderá ser definido num novo ficheiro *header* `stats.h`. Esta estrutura deverá ter os campos adequados para manter informações sobre:

- O número total de operações na tabela executadas no servidor. De cada vez que uma *thread* secundária recebe um pedido de uma operação na tabela *hash*, deverá incrementar um contador de operações realizadas após enviar a resposta ao cliente. A execução da operação `stats` (descrita na secção seguinte) não deve ser contabilizada, pois esta não realiza nenhuma operação na tabela.
- O tempo total acumulado gasto na execução de operações na tabela. As *threads* secundárias devem medir o tempo gasto a executar as operações na tabela *hash*. A medição pode ser feita usando a função `gettimeofday()`, que permite obter *timestamps* com a resolução de microssegundos. A função deve ser chamada antes e depois da execução da operação. A diferença entre os dois *timestamps* obtidos corresponde ao tempo gasto. Deve-se contabilizar o tempo total gasto, acumulado, em microssegundos.
- O número de clientes atualmente ligados ao servidor. Este número deve ser atualizado quando um novo cliente se liga ou termina uma ligação. A atualização deve ser feita pela nova *thread* no início e no fim da sua execução.

## 2.3 Operação `stats`

Adicionalmente, os alunos devem implementar a nova operação `stats`, através da qual o cliente conseguirá obter as estatísticas do servidor. Segue uma apresentação do formato das mensagens referentes à operação *stats*:

| COMANDO UTILIZADOR | MENSAGEM DE PEDIDO | MENSAGEM DE RESPOSTA |
|---|---|---|
| `stats` | `OP_STATS CT_NONE` | `OP_STATS+1 CT_STATS <stats>` ou `OP_ERROR CT_NONE` |

**Tabela 1**: Novo pedido *stats*.

Deverá ser criado um novo *opcode* na definição da mensagem no ficheiro `htmessages.proto`, da seguinte forma:

```
/* Define os possíveis opcodes da mensagem */
...
OP_STATS          70
...
```

Não esquecer de adicionar o método `stats` ao `client_stub.c/h`:

```
#ifndef _CLIENT_STUB_H
#define _CLIENT_STUB_H

...

/* Obtém as estatísticas do servidor. */

struct statistics_t *rtable_stats(struct rtable_t *rtable);

#endif
```

Como é evidente, também será necessário incluir o tratamento da operação `stats` tanto no código do cliente (`client_hashtable.c`) como no método `invoke` do `server_skeleton.c`.

## 2.4 Gestão da Concorrência

No servidor vão existir duas estruturas de dados partilhadas que poderão ser acedidas e modificadas **concorrentemente** pelas *threads*: a *hash table* e a estrutura `statistics_t` (ou as variáveis que guardam as estatísticas, caso não seja usada uma estrutura). Assim, é necessário garantir que certas operações são realizadas de forma atómica. Por exemplo, a operação `put` coloca na tabela uma entrada com o par *<chave, valor>* e altera a dimensão de uma lista interna à tabela. Se esta operação for interrompida a meio para ser executada outra operação `put` (por outra *thread*) ou mesmo uma operação `get`, a tabela pode estar num estado inconsistente e podem surgir resultados inesperados e incorretos, e a tabela pode ficar corrompida. Assim, é necessário identificar as sequências de operações que formam as secções críticas, as quais não devem ser interrompidas, e gerir a concorrência nos acessos aos dados partilhados.
Existem dois mecanismos que podem ser usados para concretizar a gestão da concorrência: *mutexes* (tipo

`pthread_mutex_t`) e variáveis condicionais (tipo `pthread_cond_t`). A forma como são utilizados é da responsabilidade dos alunos. Contudo, é importante notar que se deve permitir a execução concorrente tanto quanto possível, evitando bloquear operações que não vão comprometer a correção do sistema. Isto significa que deve ser possível executar concorrentemente duas operações que apenas leiam dados (da tabela ou de estatísticas), apenas forçando o bloqueio de *threads* no caso de existir conflito com operações de "escrita". Em suma, **sugere-se que seja seguido o modelo *single writer / multiple readers***, ou seja, as sequências de operações que alteram os dados têm de ser feitas em exclusão mútua, mas as sequências de operações de leitura podem ser feitas em paralelo sem bloqueios.

# 3 Makefile

Os alunos deverão manter o `Makefile` usado na 2ª fase do projeto, atualizando-o para compilar novo código, se necessário. Caso ainda não tenham feito, será necessário **adicionar** ao `Makefile` o seguinte alvo:

- `$(OBJ_DIR)/htmessages.pb-c.o: htmessages.proto`
  Este target define uma dependência do ficheiro objeto `htmessages.pb-c.o` relativamente ao ficheiro `htmessages.proto`. Se o ficheiro `htmessages.proto` for alterado, será necessário executar o compilador de *protobufs*, colocando depois os ficheiros resultantes nas diretorias devidas (`include` para o `.h` e `source` para o `.c`).

# 4 Entrega

A entrega da 3ª fase do projeto tem de ser feita de acordo com as seguintes regras:

1. Colocar todos os ficheiros do projeto, bem como o ficheiro `README` mencionado abaixo, num ficheiro com compressão no formato ZIP. O nome do ficheiro será **`SD-XX-projeto3.zip`** (**`XX`** é o número do grupo).
2. Submeter o ficheiro **`SD-XX-projeto3.zip`** na página da disciplina no moodle da FCUL, utilizando a atividade disponibilizada para tal. Apenas um dos elementos do grupo deve submeter, considerando-se apenas a submissão mais recente no caso de existirem várias.

O ficheiro ZIP deverá conter uma diretoria cujo nome é **`SD-XX`**, onde **`XX`** é o número do grupo. Nela serão colocados:

- o ficheiro **`README`**, onde os alunos podem incluir informações que julguem necessárias (e.g., limitações na implementação);
- diretorias adicionais, nomeadamente:
  - **`include`**: para armazenar os ficheiros `.h`;
  - **`source`**: para armazenar os ficheiros `.c`;
  - **`lib`**: para armazenar bibliotecas;
  - **`object`**: para armazenar os ficheiros objeto;
  - **`binary`**: para armazenar os ficheiros executáveis.
- um ficheiro **`Makefile`** que permita a correta compilação de todos os ficheiros entregues. Não devem ser incluídos no ficheiro ZIP os ficheiros objeto (`.o`) ou executáveis que são construídos pelo `Makefile`. Caso sejam usados os ficheiros objeto da 1ª fase do projeto disponibilizados aos grupos, estes **devem** ser incluídos no ficheiro ZIP.

Na entrega do trabalho, é ainda necessário ter em conta que:

- **Se não for incluído um `Makefile`, se o mesmo não compilar os ficheiros fonte, ou se houver erros de compilação (isto é, se não forem criados os ficheiros objeto e executáveis), o trabalho é considerado nulo**. Na página da disciplina, no Moodle, podem encontrar documentos do utilitário `make` e dos ficheiros `Makefile` (cortesia da disciplina de Sistemas Operativos).
- Todos os ficheiros entregues devem começar com um cabeçalho com três ou quatro linhas de comentários a dizer o número do grupo e o nome e número dos seus elementos.
- Os programas são testados no ambiente dos laboratórios de aulas, pelo que se recomenda que os alunos testem os seus programas neste mesmo ambiente.

**O prazo de entrega é dia 24/11/2024, até às 23:59 horas.**

Após esta data, a submissão do trabalho através do Moodle deixará de ser permitida.

# 5 Autoavaliação de contribuições

Cada aluno tem de preencher no Moodle um formulário de autoavaliação das contribuições individuais de cada elemento do grupo para o projeto. Por exemplo, se todos os elementos colaboraram de forma idêntica, bastará que todos indiquem que cada um contribuiu 33%. Aplicam-se as seguintes regras e penalizações:

- Alunos que não preencham o formulário até a data limite de entrega do projeto **sofrem uma penalização na nota de 20%.**
- Caso existam assimetrias significativas entre as respostas de cada elemento do grupo, o grupo poderá ser chamado para as explicar.
- Se as contribuições individuais forem diferentes, isso será refletido na nota de cada elemento do grupo, levando à atribuição de notas individuais diferentes.

**O prazo de preenchimento desde formulário é o mesmo que a entrega do projeto (24/11/2024, até às 23:59 horas).**

# 6 Plágios

Não é permitido aos alunos partilharem códigos com soluções, ainda que parciais, de nenhuma parte do projeto com outros alunos (nem através do Fórum da disciplina, nem por qualquer outro meio). Além disso, todos os códigos serão testados por um verificador de plágio. Caso alguma irregularidade seja encontrada, os projetos de todos os alunos envolvidos serão anulados e o caso será reportado aos órgãos responsáveis em Ciências@ULisboa.

Chamamos a atenção para o facto das plataformas generativas baseadas em Inteligência Artificial (e.g., o ChatGPT e o GitHub Co-Pilot) (1) gerarem um número limitado de soluções diferentes para o mesmo problema, (2) podem não resolver corretamente as alíneas descritas no projeto, e (3) incluem padrões característicos deste tipo de ferramenta, as quais podem vir a ser detetáveis pelos verificadores de plágio. Desta forma, recomendamos fortemente que os alunos não submetam trechos de código gerados por este tipo de ferramenta a fim de evitar riscos desnecessários.

Por fim, é responsabilidade de cada aluno garantir que a sua *home*, as suas diretorias e os seus ficheiros de código estão protegidos contra a leitura de outras pessoas (que não o utilizador dono dos mesmos). Por exemplo, se os ficheiros estiverem gravados na sua área de aluno nos servidores de Ciências@ULisboa, então todos os itens mencionados anteriormente devem ter as permissões de acesso `700`. Se os ficheiros estiverem no GitHub, garantam que o conteúdo do vosso repositório não esteja visível publicamente. Caso contrário, a sua participação num eventual plágio será considerada ativa.

# 7 Bibliografia

[1] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. Proc. of the 21st Symposium on Operating System Principles – SOSP'07. pp. 205-220. Out. de 2007.

[2] Wikipedia. Linked List. https://en.wikipedia.org/wiki/Linked_list.

[3] Wikipedia. Hash Table. http://en.wikipedia.org/wiki/Hash_table.

[4] B. W. Kernighan, D. M. Ritchie, C Programming Language, 2nd Ed, Prentice-Hall, 1988.