Projeto de Sistemas Operativos 2024-25
Enunciado da 2ª parte do projeto LEIC-A/LEIC-T/LETI

A segunda parte do projeto consiste de 2 exercícios que visam:

i) Tornar o IST-KVS acessível a processos clientes que queiram monitorizar alterações de certos pares chave-valor ligando ao IST-KVS através de *named pipes*,

ii) Terminar as ligações entre o IST-KVS e os clientes usando sinais.

Código base

O código base fornecido para esta 2ª parte contém uma implementação do servidor que corresponde a uma possível solução da primeira parte do projeto. Também contém uma implementação da API de cliente vazia. Os comandos utilizados nesta parte do projeto são todos os introduzidos na primeira parte e são estendidos de forma a permitir a monitorização de alterações ao IST-KVS.

Exercício 1. Interação com processos clientes por *named pipes*

O IST-KVS deve passar a ser um processo servidor autónomo, lançado da seguinte forma:

```
kvs dir_jobs max_threads backups_max nome_do_FIFO_de_registo
```

Quando se inicia, o servidor deve criar um *named pipe* cujo nome (*pathname*) é o indicado como 4º argumento no comando acima. É através deste *pipe* que os processos clientes se poderão ligar ao servidor e enviar pedidos de início de sessão. O 1º argumento é idêntico à primeira parte do projecto e indica a directoria onde o servidor deverá ler os ficheiros de comandos a executar. O 2º argumento é o número máximo de tarefas a processar ficheiros jobs e o 3º argumento indica o número máximo de backups concorrentes, tal como na 1ª entrega do projeto. O cliente deverá ser lançado, com:

```
client id_do_cliente nome_do_FIFO_de_registo
```

Qualquer processo cliente pode ligar-se ao *pipe* do servidor e enviar-lhe uma mensagem a solicitar o início de uma sessão. Esse pedido contém os nomes de três *named pipe*, que o cliente previamente criou para a nova sessão. É através destes *named pipes* que o cliente enviará futuros pedidos para o servidor e receberá as correspondentes respostas do servidor no âmbito da nova sessão (ver Figura 1).

O cliente deve criar o FIFO de notificações, o FIFO de pedido e o FIFO de resposta adicionando ao nome dos pipes o id único recebido como argumento, como está no código base.

O servidor aceita no máximo S sessões em simultâneo, em que S é uma constante definida no código do servidor. Isto implica que o servidor, quando recebe um novo pedido de início de sessão e tem S sessões ativas, deve bloquear, esperando que uma sessão termine para que possa criar a nova.

Durante uma sessão, um cliente pode enviar comandos para adicionar ou remover subscrições a chaves, usando os FIFOs de pedidos e de respostas. Enquanto a sessão está ativa, o cliente irá obter atualizações para todas as chaves subscritas até aquele momento através do FIFO de notificações.

Uma sessão dura até ao momento em que o servidor detecta que o cliente está indisponível, ou o servidor recebe o SIGUSR1, como descrito no Ex.3. Nas subsecções seguintes descrevemos a API cliente do IST-KVS em maior detalhe, assim como o conteúdo das mensagens de pedido e resposta trocadas entre clientes e servidor.

API cliente do IST-KVS

Para permitir que os processos cliente possam interagir com o IST-KVS, existe uma interface de programação (API), em C, a qual designamos por API cliente do IST-KVS. Esta API permite ao cliente ter programas que estabelecem uma sessão com um servidor e, durante essa sessão, adicionar e remover subscrições a pares chave-valor da tabela de dispersão gerida pelo IST-KVS. Quando um cliente subscreve uma dada chave, isso significa que, sempre que haja uma alteração do valor dessa chave, esse cliente será notificado do novo valor através de um pipe ligado ao servidor. De seguida apresentamos essa API.

As seguintes operações permitem que o cliente estabeleça e termine uma sessão com o servidor:

- `int kvs_connect (char const *req_pipe_path, char const *resp_pipe_path, char const *notifications_pipe_path, char const *server_pipe_path)`
  Estabelece uma sessão usando os *named pipes* criados.
  Os *named pipes* usados pela troca de pedidos e respostas e pela recepção das notificações (isto é, após o estabelecimento da sessão) devem ser criados pelo cliente (chamando *mkfifo)*. O *named pipe* do servidor deve já estar previamente criado pelo servidor, e o correspondente nome é passado como argumento inicial do programa.

  Retorna 0 em caso de sucesso, 1 em caso de erro.

- `int kvs_disconnect()`
  Termina uma sessão ativa, envia o pedido de disconnect para o FIFO de pedido identificada na variável respectiva do cliente, fechando os *named pipes* que o cliente tinha aberto quando a sessão foi estabelecida e apagando os *named pipes* do cliente. Deve ter o efeito no servidor de eliminar todas as subscrições de pares chave-valor deste cliente.

Retorna 0 em caso de sucesso, 1 em caso de erro.

Tendo uma sessão ativa, o cliente pode invocar as seguintes operações junto do servidor:

- `int kvs_subscribe(char const* key)`
- `int kvs_unsubscribe(char const* key)`

A operação de `kvs_subscribe` permite a um cliente indicar ao servidor que pretende acompanhar as alterações a uma chave e devolverá ao cliente através do FIFO de resposta um caráter com o valor 0 ou 1 em que 0 indica que a chave dessa não existia na hashtable e 1 indica que existia.
A operação de `kvs_unsubscribe` permite a um cliente indicar ao servidor que pretende parar de acompanhar as alterações a uma chave e devolverá ao cliente através do FIFO de resposta um caracter com o valor 0 ou 1, em que 0 indica que a subscrição existia e foi removida, e 1 que a subscrição não existia.

Cada vez que um cliente recebe uma resposta a um seu comando (connect, disconnect, subscribe, unsubscribe) pelo FIFO de resposta deve ser imprimida uma nova linha no seu stdout com uma mensagem no seguinte formato:

“Server returned <response-code> for operation: <connect|disconnect|subscribe|unsubscribe>

Para além de gerir os pedidos acima, o servidor pode enviar para o FIFO de Notificações (ver Figura 1) de qualquer cliente atualizações de chaves para as quais tenha acabado de ser efetuada uma escrita. Pretende-se que seja a tarefa que efetua uma escrita na hashtable a enviar a atualização aos clientes que sejam subscritores desse par chave-valor; na primeira parte do projeto um esta tarefa escrevia para o ficheiro .out, agora escreverá para o FIFO de Notificações.
Diferentes processos clientes podem existir concorrentemente, todos eles invocando a API acima indicada (concorrentemente entre si). Por simplificação, devem ser assumidos estes pressupostos:

- Cada processo cliente deverá usar duas tarefas. A tarefa principal lê do stdin os comandos: (SUBSCRIBE, UNSUBSCRIBE, DELAY e DISCONNECT) e gere o envio de pedidos para o servidor e a recepção das correspondentes respostas, cujo formato é indicado a seguir. Uma segunda tarefa é responsável por receber notificações e imprimir o resultado para o stdout. Para facilitar os testes da aplicação, podem usar também o comando de DELAY X para atrasar a execução em X segundos (isto é necessário pois entre os SUBSCRIBE e UNSUBSCRIBE vão possivelmente querer adicionar um DELAY). O parser para estes comandos já está disponibilizado no código base (ficheiro *parser.c*). O ficheiro test_client.txt inclui um possível teste, que poderá ser executado redireccionando o std_input do client para o ficheiro test_client.txt, como exemplificado a seguir:
  ./client/client c2 register_fifo < test_client.txt
- Os processos cliente são corretos, ou seja cumprem a especificação que é descrita no resto deste documento. Contudo, podem desconectar-se de forma imprevista e isto não deve pôr em risco a correção do processo servidor.
- Quando um cliente fecha os pipes que o ligam ao servidor e o servidor detecta esse evento, o servidor deve eliminar todas as subscrições desse cliente.

Ao receber do servidor a notificação que uma certa chave mudou, o cliente deve imprimir no seu *stdout* uma mensagem no seguinte formato:

- (<chave>,<valor>)

,em que <*chave*> é a chave alterada e <*valor*> é o novo valor dessa chave.
Cada par (<chave>,<valor>) deve aparecer no terminal numa linha separada.

Caso uma chave subscrita seja apagada, deverá ser imprimido:
- (<chave>,DELETED)

Protocolo de pedidos-respostas

O conteúdo de cada mensagem (de pedido e resposta) da API cliente deve seguir o seguinte formato:

`int kvs_connect(char const *req_pipe_path, char const* resp_pipe_path, char const *notifications_pipe_path, char const *server_pipe_path)`
Mensagens de pedido
(char) OP_CODE=1 | (char[40]) nome do pipe do cliente (para pedidos) | (char[40]) nome do pipe do cliente (para respostas) | (char[40]) nome do pipe do cliente (para notificações)
Mensagens de resposta
(char) OP_CODE=1 | (char) result

`int kvs_disconnect(void)`
Mensagens de pedido
(char) OP_CODE=2
Mensagens de resposta
(char) OP_CODE=2 | (char) result

`int kvs_subscribe(char const* key)`
Mensagens de pedido
(char) OP_CODE=3 | char[41] key
Mensagens de resposta
(char) OP_CODE=3 | (char) result

`int kvs_unsubscribe(char const* key)`
Mensagens de pedido e resposta
(char) OP_CODE=4 | (char) [41] key
Mensagens de resposta
(char) OP_CODE=4 | (char) result

Onde:

- O símbolo `|` denota a concatenação de elementos numa mensagem. Por exemplo, a mensagem de resposta associada à função `kvs_disconnect` consiste num *byte* (`char`) seguido de um outro *byte* (`char`).
- Todas as mensagens de pedido são iniciadas por um código que identifica a operação solicitada (`OP_CODE`).
- As *strings* são de tamanho fixo (40 caracteres). No caso de nomes de tamanho inferior, os caracteres adicionais devem ser preenchidos com '`\0`'.
- As mensagens que o servidor envia aos clientes através do FIFO de notificações correspondente a esse cliente cada vez que há uma actualização de uma chave deverão consistir de 2 vetores de 41 caracteres (40 para o nome da chave, 1 para o caráter de terminação do nome da chave, 40 para o valor e 1 para o caráter de terminação do valor, caso a chave ou o valor não sejam de 40 caracteres, deverá ser adicionado padding para tal).
- O caso no qual o *byte* (`char`) *result* tem valor igual a 0 indica sucesso, qualquer outro valor indica um problema na execução de *kvs_connect* ou *kvs_disconnect*.

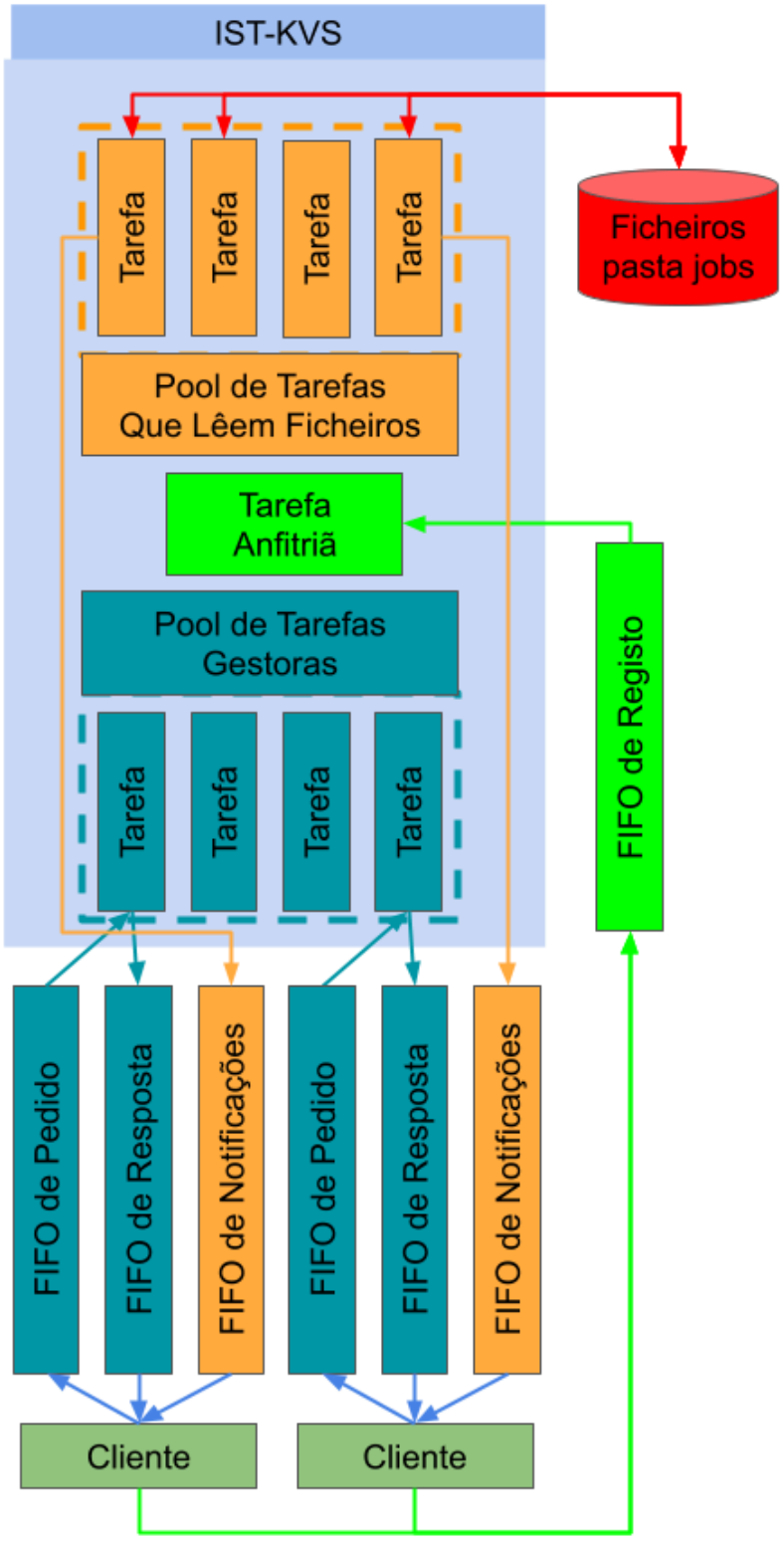

Figura 1: Arquitetura do IST-KVS (a verde o processo de registo, a azul o processo de pedido e resposta, a laranja o processo de notificações, a vermelho a leitura dos ficheiros .job).

Implementação em duas etapas

Dada a complexidade deste requisito, recomenda-se que a solução seja desenvolvida de forma gradual, em 2 etapas que descrevemos de seguida.

Etapa 1.1: Servidor IST-KVS com sessão única

Nesta fase, devem ser assumidas as seguintes simplificações (que serão eliminadas no próximo requisito):

- O servidor processa os ficheiros com os jobs como na primeira parte do projecto mas usa uma única tarefa adicional para atender apenas um cliente remoto de cada vez.
- O servidor só aceita uma sessão de cada vez (ou seja, S=1).

Experimentem:
Corra o teste disponibilizado em `jobs/test.job` sobre a sua implementação cliente-servidor do IST-KVS. Confirme que o teste termina com sucesso.

Construa e experimente testes mais elaborados que exploram diferentes funcionalidades oferecidas pelo servidor IST-KVS.

Etapa 1.2: Suporte a múltiplas sessões concorrentes

Nesta etapa, a solução composta até ao momento deve ser estendida para suportar os seguintes aspetos mais avançados.

Por um lado, o servidor deve passar a suportar múltiplas sessões ativas em simultâneo (ou seja, S>1).
Por outro lado, o servidor deve ser capaz de tratar pedidos de sessões distintas (ou seja, de clientes distintos) em paralelo, usando múltiplas tarefas (*threads)*, entre as quais:

- Uma das tarefas do servidor deve ficar responsável por receber os pedidos que chegam ao servidor através do seu *pipe*, sendo por isso chamada a *tarefa anfitriã*.
- Existem também S tarefas gestoras que gerem os pedidos de subscrição, cada uma associada a um cliente e dedicada a servir os pedidos do cliente correspondente a esta sessão. As tarefas gestoras devem ser criadas aquando da inicialização do servidor. Note-se que estas tarefas são independentes das necessárias para ler os ficheiros com comandos e executá-los.

A tarefa anfitriã coordena-se com as tarefas que gerem os pedidos de subscrição da seguinte forma:

- Quando a tarefa anfitriã recebe um pedido de estabelecimento de sessão por um cliente, a tarefa anfitriã insere o pedido num buffer produtor-consumidor. As tarefas gestoras extraem pedidos deste *buffer* e comunicam com o respectivo cliente através dos *named pipes* que o cliente terá previamente criado e comunicado junto ao pedido de estabelecimento da sessão. A sincronização do *buffer* produtor-consumidor deve basear-se em semáforos (além de *mutexes*).

Experimente:
Experimente correr os testes cliente-servidor que compôs anteriormente, mas agora lançando-os concorrentemente por 2 ou mais processos cliente.

Exercício 2. Terminação das ligações aos clientes usando sinais

Estender o IST-KVS de forma que no servidor seja redefinida a rotina de tratamento do sinal SIGUSR1. Ao receber este sinal, o (servidor) IST-KVS deve memorizar (na rotina de tratamento do sinal) que recebeu um `SIGUSR1`.
Apenas a tarefa anfitriã que recebe as ligações de registo de clientes deve escutar o `SIGUSR1`. Dado que só a tarefa anfitriã recebe o sinal, todas as tarefas de atendimento de um cliente específico devem usar a função `pthread_sigmask` para inibirem (com a opção `SIG_BLOCK`) a recepção do `SIGUSR1`.
Tendo recebido um SIGUSR1, a tarefa anfitriã eliminará todas as subscrições existentes na *hashtable* e encerrará os FIFOs de notificação e de resposta para todos os clientes, isto fará com que os clientes terminem.
As tarefas ligadas aos clientes deverão lidar com o erro associado a tentar escrever ou ler de um pipe que agora estará fechado (pois a tarefa anfitriã fechou todos estes pipes), e ao se dar este erro devem esperar novamente uma conexão do próximo cliente que se tente conectar com *kvs_connect*, querendo com isto dizer que SIGUSR1 não desliga o servidor, apenas elimina as subscrições da hashtable e desconecta todos os clientes, desassociando-os das tarefas a que estão associados, estas tarefas vão depois esperar os próximos clientes.
Quando os pipes do servidor fecharem ao receber o sinal, o cliente deverá terminar.

Ponto de partida.

Para resolver a 2ª parte do projeto, os grupos podem optar por usar como base a sua solução da 1ª parte do projeto ou aceder ao novo código base. Caso se opte por usar a solução da 1ª parte do projeto como ponto de partida, poder-se-á aproveitar a lógica de sincronização entre tarefas.

Submissão e avaliação

A submissão é feita através do Fénix **até ao dia 13 de Janeiro às 23h59**.

Os alunos devem submeter um ficheiro no formato `zip` com o código fonte e o ficheiro `Makefile`. O arquivo submetido não deve incluir outros ficheiros (tais como binários). Além disso, o comando `make clean` deve limpar todos os ficheiros resultantes da compilação do projeto.

Recomendamos que os alunos se assegurem que o projeto compila/corre corretamente no cluster *sigma*. Ao avaliar os projetos submetidos, em caso de dúvida sobre o funcionamento do código submetido, os docentes usarão o *cluster* sigma para fazer a validação final.

O uso de outros ambientes para o desenvolvimento/teste do projeto (e.g., macOS, Windows/WSL) é permitido, mas o corpo docente não dará apoio técnico a dúvidas relacionadas especificamente com esses ambientes.

A avaliação será feita de acordo com o método de avaliação descrito no site da cadeira.