Trabalho Prático de Sistemas Operativos

Grupo TPSO2

Humberto Gomes A104448

José Lopes A104541

José Matos A100612

7 de maio de 2024

Resumo

Foi proposto ao nosso grupo o desenvolvimento de um serviço de orquestração, constituído por um cliente e por um servidor responsável pelo escalonamento das tarefas submetidas pelo cliente. Neste documento, é descrita a arquitetura do *software* desenvolvido, tanto de um ponto de vista de gestão de processos, como também da organização modular do código do cliente e do servidor. Ademais, é explicado o processo de testagem feito e são enumeradas as decisões importantes tomadas para a estrutura do projeto. Conclui-se que, apesar das dificuldades encontradas no desenvolvimento, o produto entregue satisfaz os requisitos propostos. No entanto, muitas melhorias possíveis requiririam o uso de sinais e uma arquitetura do servidor completamente distinta.

1 Arquitetura multiprocesso

O servidor desenvolvido utiliza uma arquitetura multiprocesso para ser capaz de correr várias tarefas em simultâneo, tal como pedido pelo enunciado. O processo principal, o orquestrador, é responsável por criar e coordenar outros processos. O modo como o servidor responde a mensagens do cliente criando novos processos é descrito nesta secção. As mensagens mencionadas aqui são descritas detalhadamente em anexo, na especificação do protocolo utilizado.

Quando o cliente pede a execução de uma nova tarefa através de uma escrita num *pipe* com nome, o orquestrador é responsável pelo seu *parsing*, devolvendo ao cliente uma mensagem de sucesso (`S2C_TASK_ID`) ou de erro (`S2C_ERROR`). Na figura abaixo, pode observar-se o caso em que o utilizador pede ao servidor a execução de um programa único (tipo de mensagem `C2S_SEND_PROGRAM`), mas envia-lhe uma *pipeline* (que deveria estar presente numa mensagem `C2S_SEND_TASK`), resultando num erro de *parsing* comunicado por um outro *pipe* com nome, exclusivo à instância do cliente.

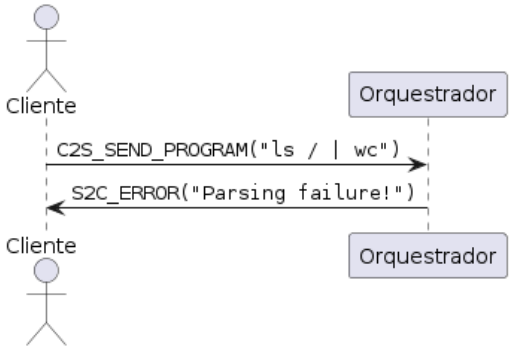

Figura 1: Comunicação entre o cliente e o servidor em caso de falha de *parsing*.

No final de cada conexão, o servidor verifica se tem capacidade para a execução de mais tarefas e, em caso afirmativo, cria um novo processo para a tarefa na frente da fila de espera. Este processo é responsável por executar os vários programas presentes na tarefa (que pode ser uma *pipeline*), esperar pelo seu término, e avisar o orquestrador que ele mesmo terminou, através de uma escrita no mesmo FIFO que o cliente utiliza (mensagem `C2S_TASK_DONE`). Deste modo, o orquestrador sabe que pode executar a *system call* bloqueante `wait()`. Na figura abaixo, observa-se a receção e escalonamento imediato da primeira tarefa recebida pelo servidor:

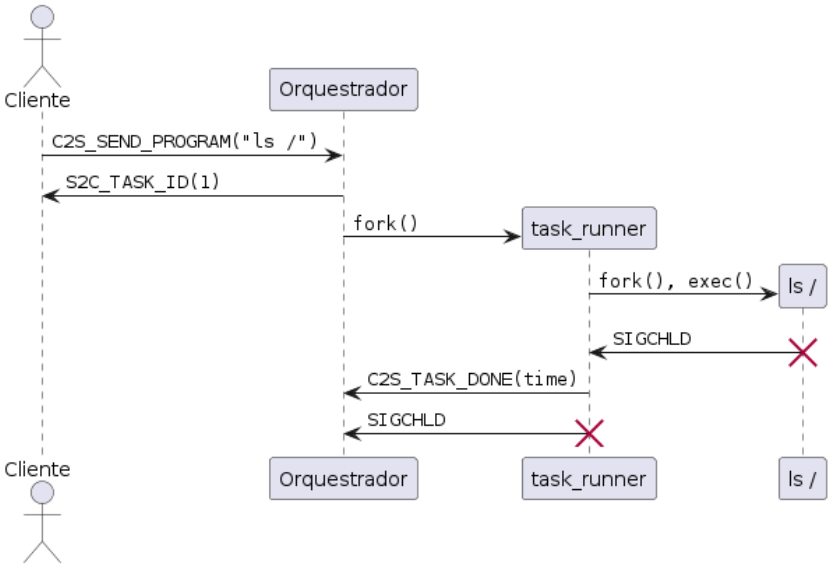

Figura 2: Comunicação entre o cliente e o servidor no caso da tarefa ser imediatamente escalonada.

O outro tipo de comunicação possível é um pedido de estado do servidor feito pelo cliente (`C2S_STATUS`). Como, para respeitar o que foi pedido pelo enunciado, o orquestrador não deve parar de ler pedidos de utilizadores enquanto comunica o seu estado a outro, a comunicação de estado ao cliente é feita por um processo distinto, como se vê na figura abaixo:

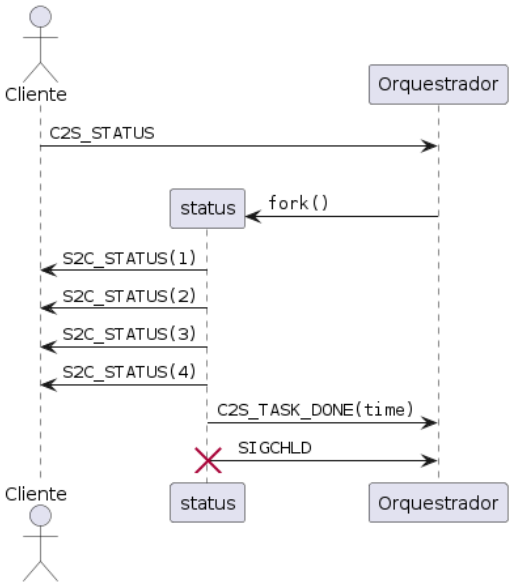

Figura 3: Comunicação entre o cliente e o servidor num pedido de estado bem sucedido.

O processo de comunicação de estado transmite ao cliente, em mensagens `S2C_STATUS`, as várias tarefas em execução, em fila de espera e já executadas (estas últimas são lidas do histórico do servidor guardado em ficheiro). Tal como ocorre com a execução de tarefas regulares, é necessário informar o orquestrador que pode aguardar pelo processo filho quase a *zombieficar-se*. Como as tarefas de estado são escalonadas do mesmo modo que as tarefas regulares, existe um limite superior de tarefas executadas em simultâneo. No entanto, para não se fazer o cliente aguardar quando não há capacidade de escalonamento disponível, é enviada uma mensagem de erro, como se vê na figura abaixo:

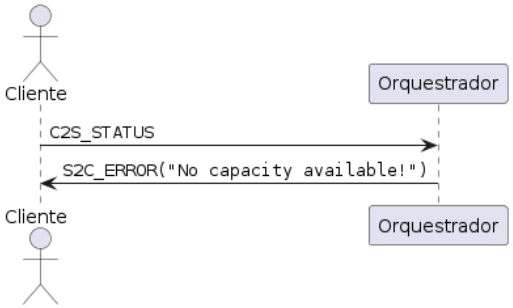

Figura 4: Comunicação entre o cliente e o servidor num pedido de estado falhado.

2 Arquitetura modular

A arquitetura do *software* desenvolvido pode ser abordada de diversas perspetivas, tal como a da organização do código em diversos módulos, vistos na figura abaixo:

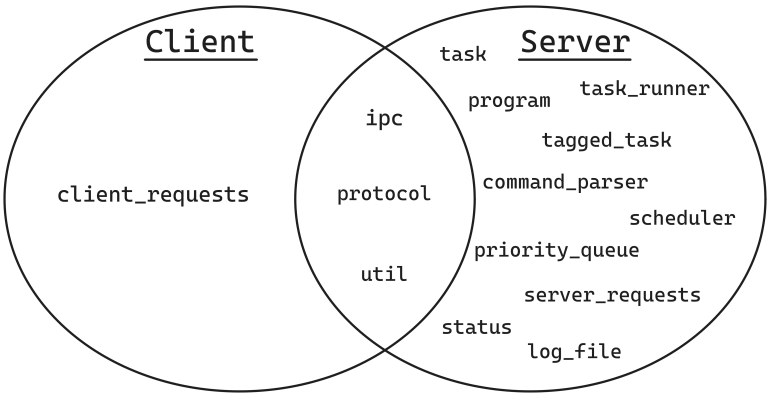

Figura 5: Módulos presentes no cliente e no servidor.

Alguns módulos são partilhados pelo cliente e pelo servidor: `util` fornece utilidades de escrita para `stdout` e `stderr` utilizando a *system call* `write`; `ipc` é responsável pela comunicação interprocesso a um nível mais básico (abertura de conexões e separação de mensagens) utilizando *pipes* com nome, enquanto que em `protocol` estão definidas as estruturas das mensagens serializadas, transmitidas entre o cliente e o servidor, e vice-versa.

O único módulo presente apenas no cliente é `client_requests`, que é responsável por enviar mensagens ao servidor e receber as suas respostas. No servidor, o módulo `server_requests` tem o papel dual: espera por pedidos do cliente para lhes responder. Este módulo é também responsável por coordenar, conforme as mensagens que são recebidas, as escritas para o ficheiro de *logs*, feitas pelo módulo `log_file`.

Há várias estruturas de dados no servidor. `program` é um *array* de argumentos terminado em `NULL`, facilmente providenciado a uma função na família `exec`. Vários programas formam uma *pipeline* em `task`, que alternativamente pode ser constituída por um procedimento de C (é possível escalonar tarefas que não programas independentes). Há mais informação necessária para a gestão de tarefas do que uma lista de programas, como identificadores e tempos em filas de espera. Uma `tagged_task` é formada por uma `task` e por esta informação. Continuando-se a cadeia de composição, tem-se a estrutura `priority_queue`, uma *minheap* de tarefas, onde definir uma política de escalonamento se torna tão simples como definir uma função de comparação entre tarefas.

Vários outros módulos são responsáveis pela funcionalidade do servidor. Apesar de uma `task` poder ser criada manualmente, programa a programa, a funcionalidade de interpretação de uma *string* de linha de comandos é providenciada pelo módulo `command_parser`. Tivemos o cuidado de que este módulo suportasse aspas únicas e duplas corretamente. Um outro módulo é `scheduler`, responsável por saber que tarefas estão em execução e em fila de espera, criando novos processos para a execução de novas tarefas desde que tenha disponibilidade para tal. Há dois programas que são executados, `status`, responsável por enviar ao cliente informação sobre o servidor sem o bloquear, e `task_runner`, responsável pela execução de tarefas como *pipelines*, informando o seu processo pai quando estas terminam.

A arquitetura modular utilizada foi essencial para o *software* desenvolvido. Não só garante código

de melhor qualidade, como permite evitar erros (como erros de memória e invariantes violadas) que ocorrem devido à quebra do encapsulamento. A manutenção do código também se tornou mais simples. Por exemplo, foi necessária, devido a um *bug* que não se conseguiu resolver, a mudança completa do protocolo definido em `ipc.c`. Foi possível reescrever este ficheiro mantendo a interface das funções nele definidas, e o servidor manteve-se operacional sem qualquer outra mudança necessária.

3 Testes executados

Foram escritos testes para avaliar diversos aspetos do *software* desenvolvido, tanto de um ponto de vista funcional como de desempenho. Nesta secção são descritos os testes desenvolvidos e as correções que deles resultaram.

O teste `fds.sh` verifica quais são os descritores de ficheiro abertos num processo criado pelo servidor em qualquer posição possível numa *pipeline*. Em princípio, o descritor 0 (`stdin`) devia estar aberto para todos os processos que não o primeiro da *pipeline*, e os descritores 1 e 2 (`stdout` e `stderr`) deveriam ser os únicos restantes abertos para qualquer processo. No entanto, outros descritores abertos foram detetados e o código de `task_runner` foi corrigido para fechar alguns descritores de *pipes* que se mantinham abertos. Este é o único teste que não corre em outros sistemas POSIX que não Linux, pois precisa de acesso à diretoria `/proc/self/fd`.

De seguida, o teste `length.sh` verifica que o servidor suporta corretamente mensagens com o comprimento máximo especificado, necessário para garantir a atomicidade de escritas no FIFO. Inicialmente, tarefas com comandos muito longos podiam ser escalonadas, mas não surgiam num pedido de `status` do cliente. Caso o comprimento do comando de uma tarefa seja superior ao permitido por uma mensagem de estado, o comando é substituído pela *string* `"COMMAND TOO LONG"`.

O teste `order.sh` garante que os processos enviados ao servidor são escalonados corretamente conforme a política de escalonamento definida. Este teste não revelou qualquer erro no comportamento do servidor. Outro teste que não revelou problemas foi `parser.sh`, que garante que o *parser* de linhas de comando funciona como esperado. Outro teste que não exigiu alterações ao código escrito foi `pipelines.sh`, que testa se os *pipes* entre comandos são capazes de transmitir mais informação do que a sua capacidade. Fá-lo através da transferência de um vídeo do YouTube, aplicando efeitos ao seu áudio, e reproduzindo-o, sem alguma vez escrever para o sistema de ficheiros.

Ademais, o teste `status.sh` procura verificar se o servidor consegue enviar o seu estado ao cliente corretamente. Detetou-se perda de mensagens quando em elevadíssimo número, erro cuja causa não se conseguiu descobrir, levando a que o módulo `ipc` fosse reescrito para deixar de usar mensagens de comprimento variável, passando a utilizar mensagens de comprimento fixo.

De seguida, `stress.sh` cria milhares de processos de clientes, que tentam comunicar com o servidor em simultâneo pelo FIFO partilhado. Detetou-se a possibilidade do servidor fechar o seu descritor durante a escrita de um cliente, resultando na terminação deste com um `SIGPIPE`. Não é um grande

problema o cliente falhar, mas esta falha de comunicação pode ocorrer quando um processo filho do orquestrador tenta avisar o seu pai que a sua execução está prestes a terminar. Se o servidor não receber esta mensagem, nunca saberá que um dos seus filhos terminou e que passa a ter mais capacidade de escalonamento disponível. Foi-nos permitido, pelo regente da UC, ignorar o sinal através da *system call* `signal`, permitindo que `write` termine com um código de erro, podendo retentar-se a comunicação com o servidor.

Por último, o ficheiro `benchmark.sh` procura analisar o desempenho das políticas de escalonamento *First Come First Served* (FCFS) e *Shortest Job First* (SJF), fazendo pedidos de cem tarefas de duração entre 300 milissegundos e 2 segundos. No final, o estado do servidor é requisitado, revelando informação sobre o tempo de espera e de execução das tarefas, que sofre uma conversão para um formato CSV, importado para uma folha de cálculo e analisado:

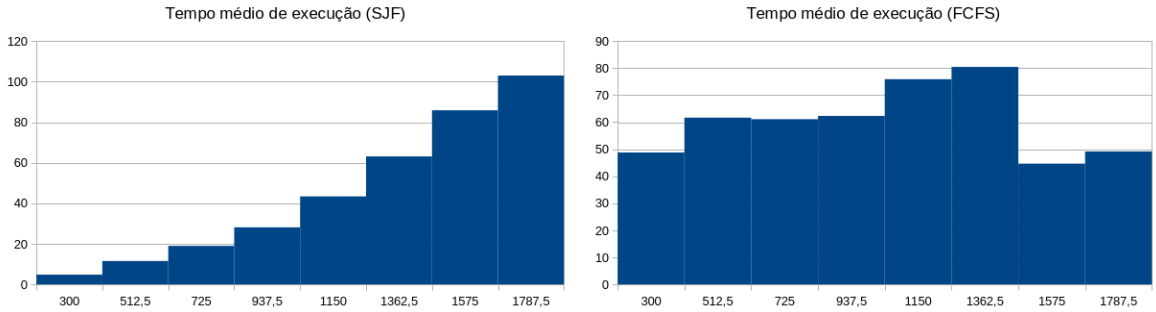

Figura 6: Tempos médios de execução de tarefas (desde o cliente até o fim da execução, em segundos) em função da sua duração (em milissegundos), agrupada em oito conjuntos, conforme a política de escalonamento usada.

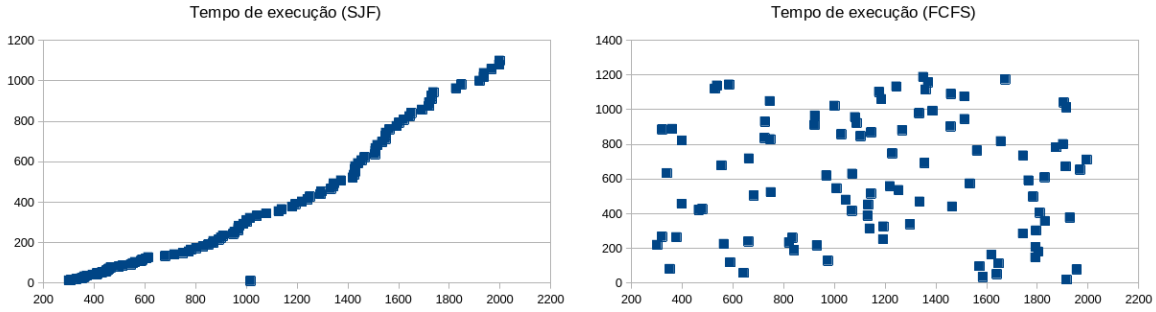

Figura 7: Tempos de execução de tarefas (desde o cliente até o fim da execução, em segundos) em função da sua duração (em milissegundos), conforme a política de escalonamento usada.

Conforme esperado, quando a política SJF é utilizada, tarefas de menor duração passam menos tempo em fila de espera (a maior contribuição para o tempo total de execução) do que as tarefas mais longas, evidenciado pelo comportamento crescente tanto no histograma como no *scatter*. Por outro lado, o tempo de execução quando se aplica a política FCFS é relativamente uniforme, evidenciado pela altura relativamente próxima das barras do histograma e da aleatoriedade no *scatter*.

Como na política SJF as tarefas mais rápidas são executadas em primeiro lugar, os tempos de espera das primeiras tarefas são muito baixos, reduzindo o tempo médio de espera total. Esta tese verifica-se:

SFJ apresenta um tempo médio de espera de 402 segundos mas estes tornam-se 592 segundos para FCFS. No entanto, em FCFS, tarefas de maior duração experienciam menor tempo médio, como se observa no primeiro par de gráficos. Logo, a escolha de uma política de escalonamento depende do tipo de tarefa que se pretende valorizar: usa-se SJF para processos de curta ou média duração, e FCFS para tarefas mais longas.

4 Decisões relevantes

Nesta secção, são discutidas algumas decisões tomadas ao longo do desenvolvimento do projeto que o nosso grupo julga serem relevantes de se mencionar.

Inicialmente, o nosso grupo pensava em fazer um orquestrador seguro, capaz de estar presente num sistema partilhado (*multiseat*). No entanto, foi rápida a conclusão de que sem virtualização (ou *containerização*) seria possível um cliente enviar uma tarefa como `pkill orchestrator`. Ademais, poderiam ser envidas várias tarefas sem fim, tirando toda a capacidade ao servidor, sem que este as possa matar devido à proibição de se utilizarem sinais. Logo, o objetivo de segurança foi descartado e o foco do grupo passou para a robustez do servidor, que foi programado de modo a resistir a erros tanto internos (como *system calls* falhadas) como externos (mensagens mal formatadas no FIFO), procurando recuperar e voltar a um estado conhecido. Ademais, durante o desenvolvimento, foi sendo executado o `valgrind` [1] para procurar erros de memória como acessos indevidos e *leaks*. Após os vários testes realizados e falhas propositadamente provocadas, julgamos que o servidor seja capaz de continuar a operar corretamente sem *crashes*.

Ademais, o nosso grupo optou por utilizar uma *minheap* (`priority_queue`) para as políticas de escalonamento SJF e FCFS, apesar de tal não conduzir ao melhor desempenho. *Minheaps* apresentam complexidades (caso médio) de $\Theta(1)$ para inserção e $\Theta(\log n)$ para remoção. Estas são ideais para SJF, mas seria possível uma remoção em tempo constante para FCFS caso se utilizasse uma fila simples. No entanto, por motivos de simplicidade, preferiu-se o uso de uma única estrutura de dados com pior desempenho do que a adição de código necessária para se suportarem duas estruturas.

Além disso, deve ser mencionado que apenas foram usadas as bibliotecas `libc`, `libm` e `librt`, todas constituintes da biblioteca POSIX de C. Não foi utilizado código de terceiros para a implementação de estruturas de dados, dado que quando a equipa docente publicou a FAQ a explicitamente autorizar o seu uso, já as estruturas de dados como *arrays* dinâmicos e filas de prioridade tinham sido implementadas pelo nosso grupo.

Por último, é importante mencionar que fomos contra a recomendação do enunciado de utilizar a *system call* `gettimeofday` para a medição de intervalos temporais, utilizando `clock_gettime` no seu lugar. `gettimeofday` é uma função depreciada desde 2008 [2], sendo que a sua substituição, `clock_gettime`, já foi introduzida em 2001, pelo que o seu uso não afetaria significativamente a portabilidade do *software* desenvolvido. Ademais, esta função, quando usada com `CLOCK_MONOTONIC`, não é influenciada por mudanças no relógio do computador (por exemplo, um ajuste do utilizador),

garantindo sempre que o valor de tempo devolvido aumenta e que não se calculam intervalos de tempo negativos. [3]

5 Conclusão

Após a conclusão deste trabalho prático, há vários aspetos do produto final que podem ser discutidos. Em primeiro lugar, o nosso grupo julga ter cumprido com correção os objetivos estipulados pelo enunciado. Ademais, encontra-se satisfeito com a qualidade do código escrito: contribuem para a sua facilidade de compreensão, modificação e extensão não só a farta documentação como também a estrutura modular.

No entanto, sentiram-se algumas dificuldades que prejudicaram a qualidade do *software* produzido. Em primeiro lugar, uma parte considerável do código dedica-se à resiliência a erros e à serialização de mensagens. Este código apresenta bastante verbosidade, distraindo o seu leitor da verdadeira funcionalidade de cada procedimento. Uma possível solução para este problema seria o uso de uma linguagem de programação que não exija muita verbosidade para estas tarefas, mas permita, tal como C, controlo direto sobre a interação com o sistema operativo. Por exemplo, Zig encaixa-se nestes critérios e o nosso grupo julga que seja uma linguagem ideal para o projeto de SO.

Ademais, devido à natureza dos descritores de ficheiros em sistemas UNIX, foi difícil conservar uma boa arquitetura modular. Como são duplicados para processos filhos, foi frequente a necessidade de estes fecharem os descritores, mas tal complexifica as relações de dependência entre módulos, dado que os processos criados precisam de conhecer todas as estruturas de dados com descritores abertos. Se o *standard* POSIX definisse uma *flag* semelhante a `O_CLOEXEC` mas para a *system call* `fork()`, não teríamos este problema.

Logo, considerando as limitações com que tivemos de lidar, o nosso grupo considera ter desenvolvido uma solução de qualidade para o problema proposto pelo enunciado. No entanto, caso fosse permitido o uso de sinais, poder-se-ia desenvolver um servidor completamente diferente, com suporte para políticas de escalonamento com desafetação forçada, *timeouts* para assegurar uma maior proteção contra ataques *Denial of Service* e muitas outras funcionalidades, apesar de tal exigir uma arquitetura muito distinta.

6 Bibliografia

[1] Valgrind Developers, "Valgrind". valgrind.org. `https://valgrind.org/` (accessed May 4, 2024).

[2] IEEE and The Open Group, "The Open Group Base Specifications Issue 7, 2018 edition". pubs.opengroup.org `https://pubs.opengroup.org/onlinepubs/9699919799/functions/gettimeofday.html` (accessed May 4, 2024).

[3] IEEE and The Open Group, "The Open Group Base Specifications Issue 7, 2018 edition". pubs.opengroup.org `https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_gettime.html` (accessed May 4, 2024).

7 Anexo – Protocolo de comunicação entre processos

7.1 Camada `ipc`

O protocolo de comunicação desenvolvido apresenta dois níveis, de um modo semelhante ao que ocorre quando datagramas IP são encapsulados em tramas Ethernet. As tramas deste nível inferior têm sempre o mesmo cabeçalho, constituído por um comprimento antes da mensagem encapsulada. Este nível protocolar permite a separação das mensagens umas das outras.

`payload_length`	`message`
`uint32_t`	`uint8_t[]`

Como a comunicação entre processos ocorre dentro do mesmo computador, a *endianess* utilizada no protocolo é a do sistema em que o *software* é utilizado. Ademais, cada trama tem um tamanho máximo de `PIPE_BUF` *bytes*, para garantir a atomicidade de escrita nos *pipes* com nome.

7.2 Camada `protocol`

As mensagens enviadas entre o servidor e o cliente são encapsuladas em tramas `ipc`. Os nomes das mensagens do cliente para o servidor têm o prefixo `C2S`, enquanto que as mensagens em sentido contrário são identificadas por nomes com o prefixo `S2C`.

7.2.1 `C2S_SEND_PROGRAM` e `C2S_SEND_TASK`

Mensagens destes tipos têm como função a submissão ao servidor de um programa único ou de uma *pipeline*, respetivamente. Ambos os tipos possuem uma parametrização idêntica, sendo compostos por 5 campos: o tipo de mensagem, que será uma das duas opções apresentadas anteriormente (valores inteiros `0` e `1`, respetivamente); o PID do processo que enviou a mensagem; uma marca temporal do momento em que a mensagem foi enviada (para cálculo do tempo na fila do FIFO); o tempo de execução previsto pelo cliente, em milissegundos; e, por fim, o comando introduzido pelo utilizador ao executar o cliente.

type	client_pid	time_sent	expected_time	command_line
uint8_t	pid_t	struct timespec	uint32_t	char[]

Note-se que `command_line` não é terminada em `'\0'` e que o seu comprimento é determinado pelo comprimento da mensagem, definido pela camada `ipc`. O mesmo é verdade para todas as *strings* neste protocolo.

7.2.2 C2S_TASK_DONE

As mensagens do tipo `C2S_TASK_DONE` (2) são responsáveis por avisar o servidor do término da execução de uma tarefa realizada por um processo filho, que neste contexto é tratado como um cliente. A estrutura de mensagens deste tipo é caracterizada por 5 campos: o tipo da mensagem, obrigatoriamente `C2S_TASK_DONE`; a posição do escalonador em que a tarefa foi agendada (informação necessária para este saber que tarefa deve ser dada como terminada); uma marca temporal de quando a tarefa terminou a sua execução (para cálculo do tempo de execução); e, por fim, duas etiquetas: a primeira informa se a tarefa executada é, ou não, um pedido de estado do servidor (para se saber que escalonador deve terminar a tarefa), e a segunda verifica se a execução da tarefa resultou, ou não, num erro.

type	slot	time_ended	is_status	error
uint8_t	size_t	struct timespec	uint8_t	uint8_t

7.2.3 C2S_STATUS

O último tipo de mensagem enviada para o servidor é `C2S_STATUS` (3), responsável por transportar pedidos de estado. A sua estrutura é bastante simples, sendo apenas composta por 2 campos: o identificador do tipo de mensagem, obrigatoriamente `C2S_STATUS`, e o PID do cliente, para se poder responder-lhe por um outro FIFO cujo nome é determinado pelo PID do processo.

type	client_pid
uint8_t	pid_t

7.3 S2C_ERROR

Uma possível mensagem vinda do servidor é `S2C_ERROR`, responsável por avisar o cliente de erros ocorridos durante o processamento de um pedido. Apenas é necessário transportar na mensagem o identificador do tipo de mensagem (`S2C_ERROR`, de valor numérico `0`) e a *string* do erro em si.

type	error
uint8_t	char[]

Optou-se por se utilizar uma *string* para erros (em vez de um `enum`) de modo a se simplificar o programa, dado que qualquer mensagem pode ser enviada sem adições ao conjunto de erros possíveis. No entanto, o nosso grupo encontra-se ciente que isto não seria possível num programa num contexto não académico, onde funcionalidade como localização seria necessária.

7.4 S2C_TASK_ID

Mensagens do tipo `S2C_TASK_ID` (valor `1`) são respostas a mensagens `C2S_SEND_(PROGRAM|TASK)`. Estão encarregues de informar clientes que uma tarefa submetida foi registada com sucesso, devolvendo o seu identificador. Estas mensagens apresentam uma estrutura simples, contendo apenas o identificador do tipo de mensagem, obrigatoriamente `S2C_TASK_ID`, e o identificador da tarefa registada pelo servidor.

type	id
uint8_t	uint32_t

7.5 S2C_STATUS

Por fim, resta descrever o tipo de mensagem `S2C_STATUS` (`2`), cuja função é informar o cliente acerca do estado de uma tarefa em particular. São necessárias várias destas mensagens para informar o cliente do estado completo do servidor. A estrutura de mensagens deste tipo é a mais complexa, sendo cada mensagem composta por 9 campos: o tipo de mensagem, obrigatoriamente `S2C_STATUS`; o estado da tarefa (concluída, em execução, em fila de espera, com os valores numéricos 0, 1, e 2, respetivamente); o identificador da tarefa a que a mensagem se refere; uma etiqueta que identifica se um erro foi detetado durante a execução da tarefa; quatro valores temporais em microssegundos, relativos ao tempo no FIFO do cliente para o servidor, ao tempo em fila de espera, ao tempo de execução, e ao tempo no FIFO entre o filho do orquestrador e o seu processo pai; e, finalmente, a linha de comando da tarefa submetida.

type	status	id	error	time_c2s_fifo	time_waiting	time_executing	time_s2s_fifo	command_line
uint8_t	uint8_t	uint32_t	uint8_t	double	double	double	double	char[]