Relatório do Projeto
Programação de Sistemas
2.º Sem. 2017/18

Clipboard Distribuído

Inês Gaspar
n.º 84074

Pedro Moreira
n.º 85228

Grupo n.º 39

Docentes: João Silva e Ricardo Martins

9 de Junho de 2018

Conteúdo

1 Introdução

Este projeto consiste no desenvolvimento de uma aplicação de armazenamento de mensagens (de tipo indefinido) num sistema distribuído de servidores - clipboards - que visa realizar o armazenamento das mensagens de forma redundante, isto é, todas as mensagens enviadas para um clipboard são replicadas para todos os outros que fizerem parte do sistema distribuído.

É fornecida aos clientes/utilizadores uma API que lhes permite enviar ou pedir mensagens a um clipboard local, onde este pretende sempre dar uma resposta que reflita (no limite) a informação de todo o sistema distribuído de clipboards.

O projeto baseia-se numa arquitetura multi-threaded, pois cada clipboard pode comunicar, paralelamente e em simultâneo, com várias apps (de utilizadores) e com os clipboards que lhe sejam adjacentes na rede distribuída.

No desenvolvimento deste projeto foi tida em especial atenção a correção de erros, dos quais são exemplo: mensagens incoerentes vindas do utilizador, "morte"de um servidor inserido na rede distribuída, etc. É de realçar, que este foi um dos aspetos aos quais se deu mais ênfase na projeção e desenvolvimento deste trabalho.

2 Arquitetura do sistema

O método utilizado para os clipboards armazenarem as mensagens dos utilizadores foi através de regiões distintas - foi definido que cada clipboard dispõe de uma memória de 10 regiões e, desta forma, os utilizadores podem pedir (ou enviar) mensagens de (ou para) regiões diferentes sempre que válidas para o intervalo de regiões existentes. A propagação das mensagens por toda a rede é também feita de forma a preservar o sincronismo das mensagens e das respetivas regiões.

Qualquer clipboard ativo tem de estar apto a receber ligações, não só de aplicações como de outros clipboards, sendo este segundo processo a forma como se gera o sistema de servidores distribuído. Já no lançamento de cada clipboard existem duas hipóteses possíveis: modo *connected* - opção de ser lançado ligando-se a um outro clipboard já existente, e neste caso passa a fazer parte da rede onde o clipboard a que se ligou está inserido; modo *single* - opção de ser lançado sem que se conecte a nenhum clipboard, e neste caso já nunca poderá por "iniciativa própria"ligar-se a outro clipboard no decorrer do programa. A hipótese pretendida é indicada através dos argumentos de entrada do programa (pelo método definido no enunciado do projeto).

Para ser mais fácil de expor o funcionamento detalhado do programa, vão ser definidos neste parágrafo alguns conceitos que serão recorrentes no presente relatório.
Na Figura 1 está representado o esquema em árvore no qual se baseou o pensamento para retratar as ligações e fluxo de informação na rede de clipboards. Para analisar o programa onde é corrido o clipboard #4, vai chamar-se "clipboard servidor"ao clipboard ao qual se ligou quando foi lançado, que é o clipboard #2. Os clipboards #5 e #6, foram por sua vez lançados ligando-se ao clipboard #4, pelo que deste ponto de vista vão chamar-

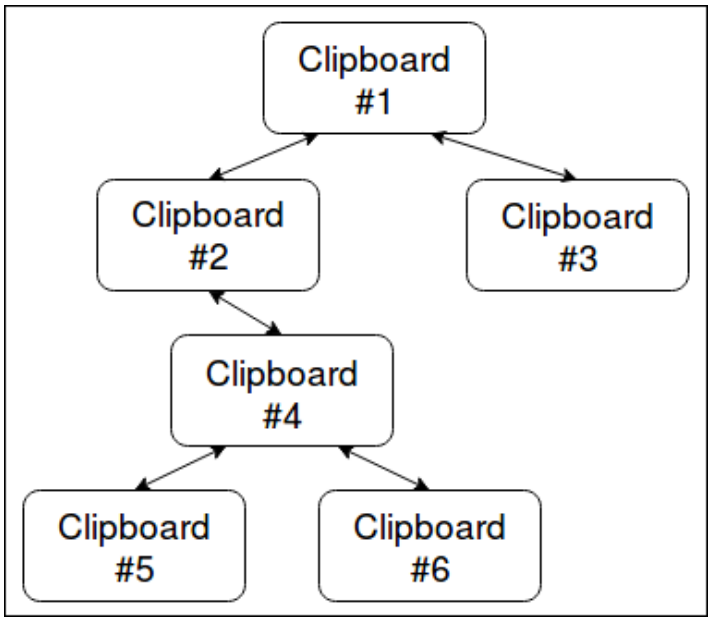

Figura 1: Esquema ilustrativo da rede distribuída

se de "clipboards clientes". O clipboard em questão (#4) vai-se designar por clipboard local (esta representação vai também na verdade relfetir o processo implementado para o fluxo de tratamento de pedidos).
À memória do clipboard que guarda as mensagens, vai chamar-se de regiões.
Um clipboard local dispõe então de 3 tipos de ligações: tipo 1 - aplicação de um utilizador; tipo 2 - clipboard cliente; tipo 3 - clipboard servidor.

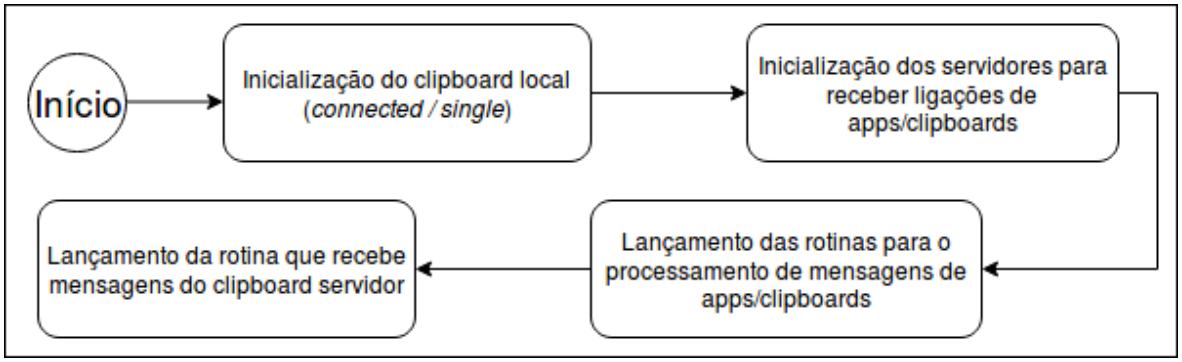

Figura 2: clipboard.c

Na Figura 2 mostra-se a estrutura geral e básica do programa, que está explícita no ficheiro clipboard.c (main). Assim que o programa é executado as regiões são inicializadas: para o modo *single* - definidas como vazias; para o modo connected - são "descarregadas"as regiões do clipboard ao qual se liga. São depois inicializados em paralelo, pois são independentes, os servidores para receberem ligações de aplicações e clipboards clientes. Com todos os "set-ups"já concluídos, lançam-se então, também em paralelo, as rotinas que processam as comunicações com aplicações, clipboards clientes e clipboard servidor. A partir deste momento, todo o decorrer do programa vai ser em função das comunicações recebidas por estas ligações.

2.1 Protocolo de Comunicação

2.1.1 Tipo de comunicação utilizada

A troca de informação nos três tipos de ligação foi efetuada através do envio de mensagens por *stream sockets*, sendo que para o tipo 1 foi utilizado o domínio UNIX e para os tipos 2 e 3 o domínio INET, para que se possa estender a rede distribuída a várias máquinas. A escolha de stream sockets face à possibilidade de datagram, foi feita com base na preferência de uma maior garantia de que não haveria perda de pacotes ou desordem dos mesmos contra uma maior velocidade de estabelecer ligações.

É relevante referir já que foram implementados dois tipos de comunicação:

- transição da mensagen - envio/receção de um array de tipo indefinido.
- transição das informações da mensagem - envio/receção de uma estrutura serializada que contém os vários indicadores necessários para que a mensagem seja transitada e posteriormente processada (definida no capítulo 3).

2.1.2 Tratamento de pedidos

Tal como anteriormente mencionado, a API disponibiliza 3 funcionalidades ativas a aplicar a um clipboard - $Copy$, $Paste$ ou $Wait$. O protocolo definido para estas ações é o seguinte: primeiro é feita uma comunicação, para enviar todas as informações relativas à mensagem pedida/a enviar - região em causa, tamanho e ordem - sendo que a ordem define a funcionalidade, e o tamanho define, no caso de mensagem pedida, o tamanho máximo a ser recebido, no caso de mensagem a enviar, o tamanho desta. No clipboard, esta informação é recebida e, após processada, há uma decisão consoante a ordem:

- O $Copy$ pretende enviar uma mensagem para uma dada região do clipboard a que se está ligado. Assim sendo, quando no clipboard é recebida a informação de que vai ser feito um $Copy$, este prepara-se para ler a mensagem com o tamanho indicado, e armazená-la na região pedida. Após o envio correto da mensagem, a função da API retorna o tamanho da mensagem enviada (em bytes).
- O $Paste$ pede a mensagem armazenada numa dada região do clipboard a que se está ligado. Assim sendo, quando no clipboard é recebida a informação de que vai ser feito um $Paste$, este verifica se a região não está vazia ou se a mensagem lá armazenada excede o tamanho máximo a ser recebido. De seguida, o clipboard informa o cliente, que ficou à espera de receber a verificação, se estes parâmetros foram cumpridos. Se falhar a verificação, ambas as partes terminam o ciclo, e a função da API retorna 0, que indica que a ação não foi efetuada. Caso a verificação suceda, o clipboard envia a mensagem pedida para o outro lado da API, que por sua vez ficou à espera desta quando recebeu a informação de que a verificação foi sucedida. Neste caso a API retorna o tamanho da mensagem recebida (em bytes).
- O $Wait$ tem a mesma função que o paste, mas só pretende receber a mensagem após haver uma alteração na região em questão, contando a partir do momento em que o $Wait$ é efetuado. Assim sendo, quando no clipboard é recebida a informação de que vai ser feito um $Wait$, espera-se que a região em questão seja alterada, e segue-se então todo o processo referido para a função $Paste$.

Após a receção/envio da mensagem no clipboard, este prepara-se receber um novo pedido, funcionando assim de forma cíclica até que se feche a ligação.
Todo este processo está devidamente expresso no fluxograma da Figura 3.

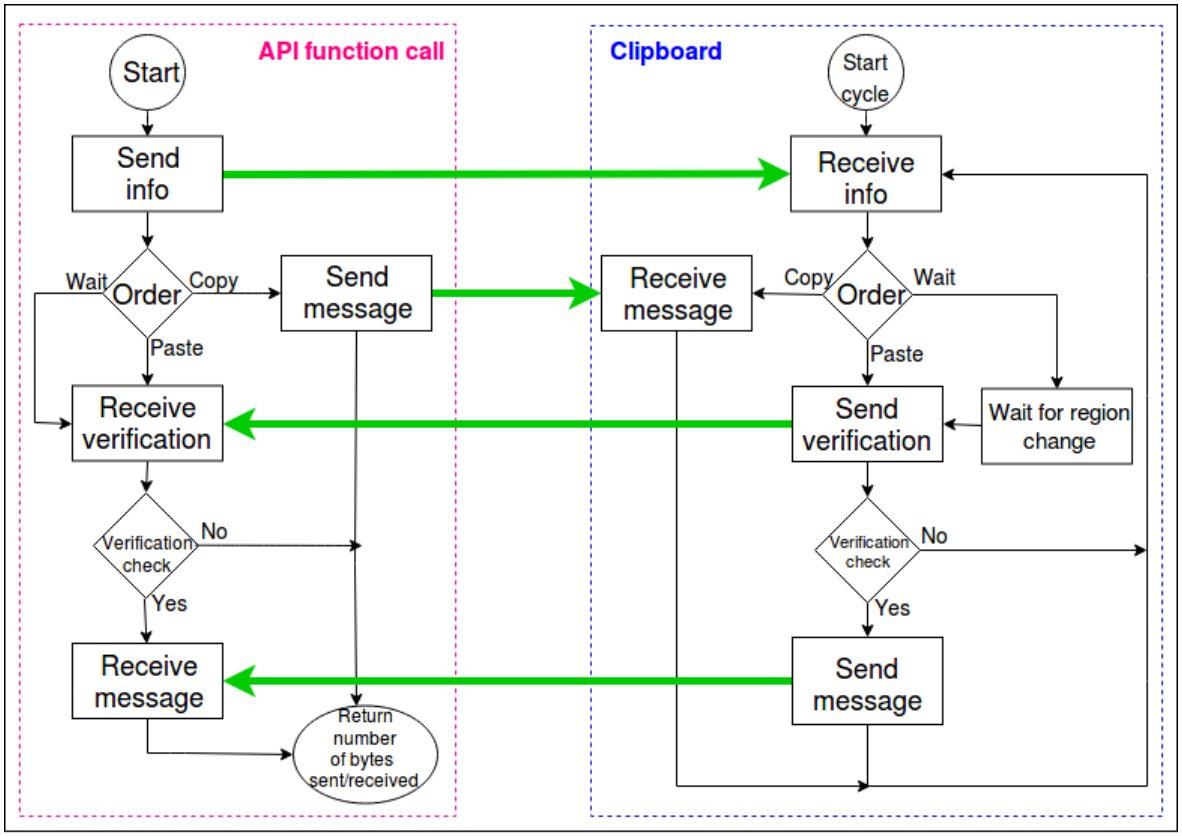

Figura 3: Fluxograma - tratamento de pedidos

Deve também ser referido, que as funções da API retornam diretamente o valor de erro (0) caso se tente utilizar umas destas três funcionalidades com: ponto de ligação ao clipboard inválido, mensagem (ou memória para a armazenar) vazia, ou também se a região requisitada for inválida (fora do intervalo de regiões que o clipboard possui).

Num âmbito de prevenção de uma possível aplicação pirata que conheça parte o protocolo e possa tentar dessintonizá-lo com uma API alterada, do lado do clipboard é sempre feita a verificação de: ordem válida (uma das 3 referidas), e refeita a verificação de região válida (que no ciclo esperado já é feita na função da API). Caso uma destas validações falhe, o clipboard fecha esta ligação e deixam de ser aceites pedidos deste cliente.

2.2 Fluxo de mensagens na rede distribuída

2.2.1 Implementação do armazenamento redundante

Olhando para a rede distribuída com uma disposição em árvore, conforme apresentado na Figura 1, a rotina que processa as ligações com o clipboard local ($connection\_handle()$ - threads.c), não faz distinção entre ligações de tipo 1 e 2 - vindas de baixo -, apenas diferencia as de tipo 3 - vindas de cima - (facto que será explicado mais à frente nesta secção) e desta forma, um $Copy$ vindo de uma ligação do tipo 1 é interpretado da mesma forma

que um *Copy* vindo de uma ligação do tipo 2.

Desde o momento em que o *Copy* é executado até ao momento em que a mensagem é efetivada nas regiões dos clipboards da rede distribuída, há duas fases de processamento desta ação:

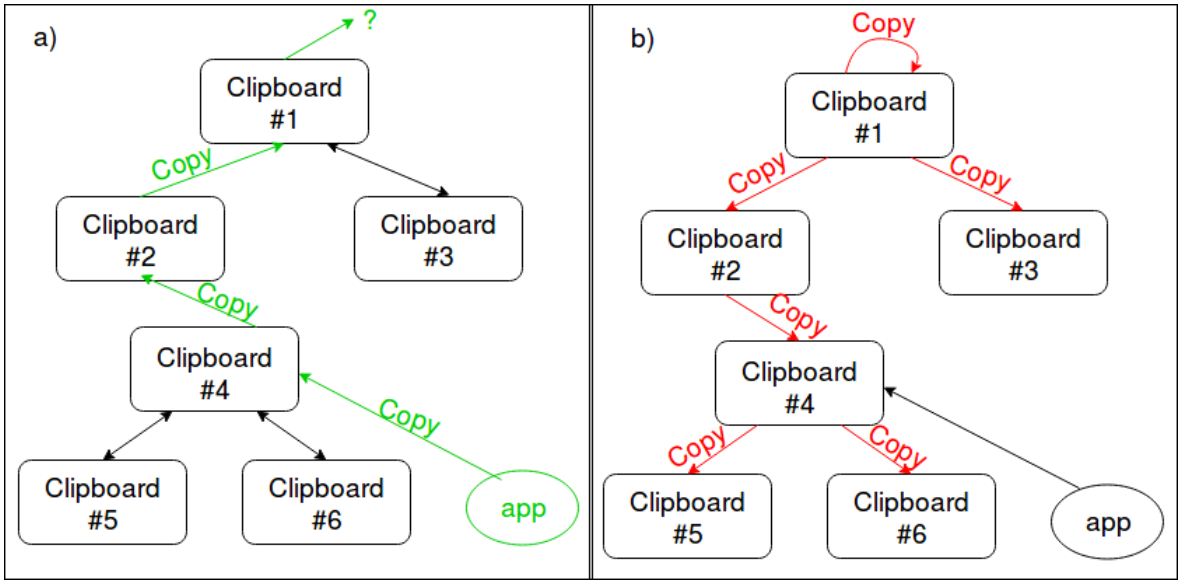

Figura 4: a) *Copy* no sentido ascendente; b) *Copy* no sentido descendente

Sentido ascendente - quando é então feito um *Copy* vindo de baixo, o clipboard local reencaminha a mensagem para o clipboard servidor e não atualiza a região pedida com a mensagem. O processo vai repetir-se até que a mensagem atinja o topo da árvore. Vamos ilustrar um exemplo prático de um *Copy* ordenado por uma aplicação, para que de forma análoga, seja explicativo para qualquer caso deste tipo:
Supondo então que ocorre uma ligação do tipo 1 ao clipboard #4 da Figura 1, a mensagem vai ser reencaminhada até que chegue ao topo da árvore (clipboard #1), como é facilmente visível na Figura 4 a).

Atingindo o topo da árvore, a mensagem chega a um clipboard que trabalha em modo *single*. O método implementado para este modo, foi através da simulação de um clipboard servidor por meio de um *pipe*, isto é, o clipboard local efetua as mesmas comunicações que em modo *connected* efetuaria com o clipboard servidor, mas escreve para o *endpoint* de escrita do *pipe* e lê do *endpoint* de leitura do mesmo. Deste modo, quando é feito um *Copy* vindo de baixo num clipboard a trabalhar em modo *single*, ele reencaminha a mensagem para si próprio, mas desta vez recebe um *Copy* vindo de cima, pois é interpretado como vindo do clipboard servidor. Com esta implementação não só se consegue garantir que só é efetivada uma mensagem de cada vez, como se tem a facilidade de poder utilizar o mesmo código para os dois modos de trabalho dos clipboards.

Sentido descendente - Quando é então feito um *Copy* vindo de cima, o clipboard local atualiza a região em questão, e efetua o equivalente a um *Copy* (diferença para o *Copy* regular explicitada na capítulo 5) para todos os clipboards clientes a si ligados. O processo vai repetir-se por cada ramo da árvore até que atinja o último andar do respetivo

ramo. Está-se agora em condições de compreender a conclusão do exemplo deixado por acabar na Figura 4 a), através da análise da Figura 4 b).

2.2.2 Gestão de threads

Conforme referido na introdução deste trabalho, a arquitetura do sistema tem um funcionamento multi-threaded. Nesta secção vai ser explicado como é feita a gestão das múltiplas threads que compõem cada programa.

O clipboard local dispõe continuamente de uma thread por ligação, para qualquer dos 3 tipos de ligações (pelo que cada ligação da Figura 1 representa também uma thread). Desta forma, para além dos recursos poupados, previnem-se vários problemas de sincronismo (que serão ilustrados no capítulo 4) que poderiam surgir com o uso excessivo de threads por ligação.

Tendo isto em conta e sabendo que o fluxo de mensagens na rede é realizado em duas fases, vai ser agora exposto o comportamento de cada thread ativa num clipboard local:

- No sentido ascendente, a função do clipboard local é receber uma mensagem de baixo e reencaminhá-la para o seu servidor (real ou simulado). A implementação desta fase foi feita de modo a que a thread que recebe a mensagem se encarrega também de a reencaminhar para cima. Na Figura 5 está ilustrado um exemplo de um clipboard que recebe dois $Copy$ (passos 1 e 2 para o primeiro, passos 3 e 4 para o segundo) vindos de dois clientes diferentes (tipo 1 ou tipo 2), e a thread que está a processar o pedido representada a vermelho.

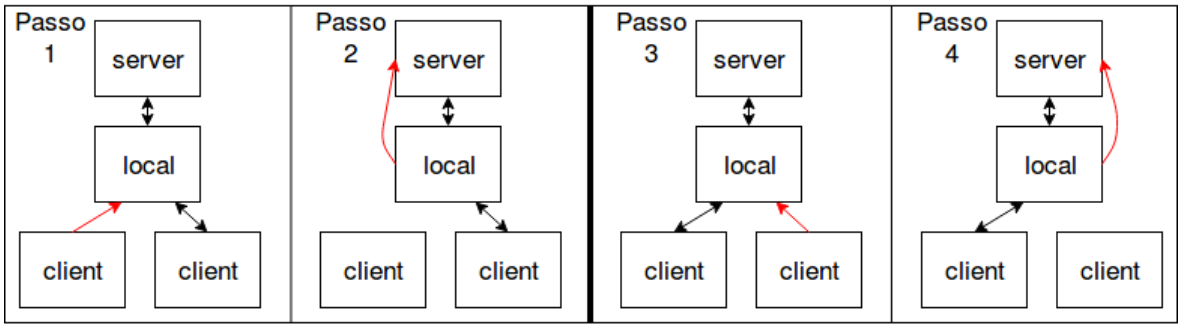

Figura 5: Manuseamento de threads no sentido ascendente

- No sentido descendente, a função do clipboard local é receber uma mensagem do clipboard servidor (real ou simulado) e reencaminhá-la para os clipboars clientes a si ligados. Para esta fase, a implementação foi feita de modo a que a thread que recebe a mensagem do clipboard servidor se encarrega também de a distribuir para os clipboards de clientes - aqui a distribuição é feita sequencialmente, correndo uma lista ligada que contém o ponto de ligação a cada clipboard cliente no nó respetivo (estrutura da lista definida no capítulo 3). Na Figura 6 está ilustrado um exemplo de um clipboard que tem três clipboards clientes a si ligados, e recebe um $Copy$ do clipboard servidor. Estão explícitas na Figura 6 todas as threads a cargo do clipboard local para as ligações definidas, bem como o seu funcionamento. A thread que processa este pedido está também representada a vermelho.

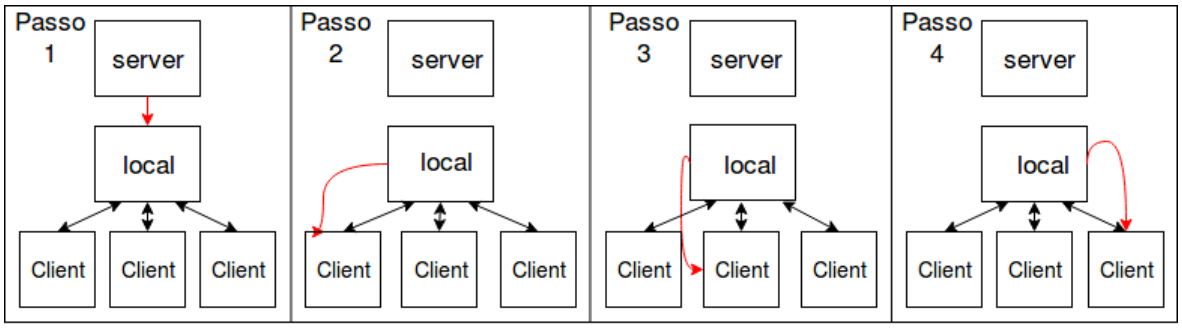

Figura 6: Manuseamento de threads no sentido descendente

3 Estruturas de Dados e Variáveis de Relevo

Na realização deste projeto foram utilizadas estruturas de dados serializadas e algumas variáveis globais cruciais para a implementação definida. Esta secção cinge-se a explicitar a função de cada uma delas.

Estruturas de dados:

- $client\_socket$ (Figura 7) - vai guardar a informação necessária para o servidor aceitar ligações de clientes. "sock_fd"tem o file descriptor com o qual já foram feitas as inicializações requeridas (funções $bind()$ e $listen()$) e "family"define o tipo de stream que vai ser criada (UNIX ou INET).

```
typedef struct client_socket{
    int sock_fd;
    int family;
}client_socket;
```

Figura 7: declaração da estrutura $client\_socket$

- $down\_list$ (Figura 8) - estrutura da lista ligada que contém em cada nó o ponto de ligação para cada clipboard cliente ligado ao clipboard local.

```
typedef struct down_list_{
    int fd;
    struct down_list_ *next;
}down_list;
```

Figura 8: declaração da estrutura $down\_list$

- $Smessage$ (Figura 9) - estrutura já referida no capítulo 2.1, que antecede a receção/envio de qualquer mensagem. "region"define a região de/para onde se pede/envia a mensagem; "message_size"define, no caso de mensagem pedida, o tamanho máximo a ser recebido, no caso de mensagem a enviar, o tamanho desta; "order"define a função a ser executada ($Copy$, $Paste$ ou $Wait$).

- REG (Figura 10) - estrutura utilizada para guardar uma mensagem no clipboard. "size"tem o valor em bytes do tamanho da mensagem; "message"é um ponteiro que aponta para a mensagem. Cada clipboard tem um vetor estático, declarado

```c
typedef struct Smessage{
    int region;
    size_t message_size;
    int order;
}Smessage;
```

Figura 9: declaração da estrutura $Smessage$

```c
typedef struct REG{
    size_t size;
    void *message;
}REG;
```

Figura 10: declaração da estrutura REG

com este tipo de estrutura, que possui 10 posições (numeradas de 0 a 9) que permitem o acesso direto tanto ao tamanho da mensagem armazenada, como à mensagem em si, para as 10 regiões que o clipboard disponibiliza.

Variáveis globais(Figura 11):

- $server\_fd$ - ponto de ligação para o clipboard servidor;
- $mutex\_init$ - bloqueia o acesso à lista dos clipboards clientes;
- $mutex\_writeUP$ - bloqueia o reencaminhamento de um copy vindo de baixo;
- $regions\_lock\_rw$ - bloqueia/seleciona o acesso às regiões do clipboard local;
- $wait\_mutexes$ - mutex necessário para implementar a variável de condição;
- $wait\_conditions$ - permite uma espera não ativa por uma alteração nas regiões do clipboard local (necessária para o $Wait$ da API).

```c
int server_fd;
pthread_mutex_t mutex_init;
pthread_mutex_t mutex_writeUP;
pthread_rwlock_t regions_lock_rw[REGIONS_NR];
pthread_mutex_t wait_mutexes[REGIONS_NR];
pthread_cond_t wait_conditions[REGIONS_NR];
```

Figura 11: Declaração das variáveis globais

4 Sincronização

Apesar da arquitetura multi-threaded utilizada ser bastante benéfica na eficiência do fluxo de tratamento de pedidos, por outro lado vai levar a alguns problemas de sincronismo. Devido à existência de várias threads a partilhar memória e com a possibilidade de lhe aceder "simultaneamente", podem ser gerados conflitos. Neste capítulo vai ser evidenciado como foram evitados estes conflitos.

4.1 Implementação da região crítica

As regiões do clipboard local podem ser requisitadas por várias threads. Quando é recebido um $Paste$ ou um $Wait$, uma região é lida, quando é recebido um $Copy$ de cima, uma região é alterada. Sendo que o processo que altera a região pode passar por eliminar a mensagem que lá estava antes e só depois atualizá-la com a nova, se entre estes dois passos houvesse uma leitura da mesma região por parte de outra thread, uma mensagem inexistente iria "ser lida"quando o expectável seria uma mensagem com o tamanho definido na estrutura que a contém.
Para resolver este conflito recorreu-se à variável $regions\_lock\_rw$ que tem dois modos de utilização:

- bloquear todos os acessos;
- bloquear os acessos de escrita na memória.

Com estes recursos, basta que sejam bloqueados os acessos de escrita na memória quando estão a haver leituras, e todos os acessos quando estão a haver escritas. Assim, o método implementado passou apenas por bloquear os acessos que visem alterar a região, no segmento em que o processamento de um $Paste/Wait$ lê a região, e bloquear todos os acessos quando, após um $Copy$ vindo de cima, está a ser alterada a região.

No seguimento da alteração de uma região no clipboard local, esta é enviada para todos os clipboards clientes, pelo que para cada envio há uma leitura da região. No entanto, é aqui visível uma das vantagens da implementação com um thread por ligação pois, sendo a thread que faz o envio da mensagem para os clipboards clientes a mesma que recebe mensagens do clipboard servidor, é também a mesma (e única) que efetivamente altera regiões, nunca havendo colisão destes dois acessos.

Faz sentido referir ainda nesta secção que quando é aceite uma nova ligação do tipo 2, o clipboard local inicializa as regiões do novo clipboard cliente fazendo um $Copy$ de cada uma das suas regiões, para as respetivas deste. Para este procedimento, as regiões do clipboard local têm que ser lidas. Bloqueia-se então também a região que está a ser enviada para acessos que visem alterar o conteúdo da região.

4.2 Implementação das exclusões mútuas

Na eventualidade de serem recebidos dois $Copy$ vindos de baixo em simultâneo, e chama-se à atenção que cada um deles é processado por uma thread diferente, ambos os pedidos vão ser reencaminhados para o clipboard servidor. Se antes do protocolo do primeiro reencaminhamento estar concluído, o segundo reencaminhamento efetua uma comunicação, vai haver uma intercalação no envio de mensagens e o protocolo vai ser violado, dessintonizando-se assim as ligações. Este contratempo foi precavido com o uso da variável $mutex\_writeUP$, que para além de bloquear a comunicação com o clipboard servidor aquando do começo prévio de outra comunicação com este, também só permite que seja iniciada uma comunicação de cada vez. Olhando para a Figura 5, garantimos com esta implementação que o passo 2 nunca coincide com o passo 4.

Para introduzir este parágrafo, vai só referir-se que, a título de evitar preocupações adicionais com sincronismos na lista ligada de clipboards clientes, a remoção de um ou vários elementos da lista só é efetuada quando a mesma está a ser corrida e é detetado que a ligação já foi fechada (ou se no momento houver uma falha de ligação).
Cada vez que é aceite uma nova ligação do tipo 2 (uma nova thread fica ativa), é criado um novo nó na lista ligada de clipboards clientes para ser lá adicionado o novo ponto de ligação. Fazendo o paralelo desta ação com a de correr a lista aquando da alteração de uma região no clipboard local, sendo estas duas tarefas desempenhadas também por threads diferentes, ocorrendo ao mesmo tempo, pode ter-se uma thread a correr a lista e outra a alterá-la, o que pode resultar numa transição incorreta ou na perda da referência do primeiro nó da lista numa das threads.
De forma a impedir esta ocorrência, deu-se uso à variável $mutex\_init$, que enquanto uma thread estiver a correr/alterar a lista, mais nenhuma pode fazê-lo até que esta acabe.

Considera-se de extrema relevância referir que, dentro da região bloqueada para adicionar à lista o novo clipboard cliente, também está inserida a função que inicializa as regiões do mesmo. Se assim não fosse, haveria a possibilidade de haver a alteração de uma região no clipboard local após a inicialização das regiões do clipboard cliente mas antes da sua inserção na lista. Por conseguinte, a região alterada não iria ser replicada para o clipboard cliente, e perder-se-ia (temporariamente ou não) o armazenamento redundante que foi entendido como sendo a "essência"deste trabalho.

4.3 Implementação da espera não ativa para o $Wait$ da API

Para implementar o processamento da função $Wait$ da API de forma exequível, torna-se incontornável que a espera pela alteração da região em questão não seja feita de forma ativa. Neste âmbito, recorreu-se à variável de condição $wait\_conditions$, que permite bloquear a execução de uma ou múltiplas threads, até que seja lançado um sinal que as liberte. Assim sendo, quando é recebida, no clipboard local, a informação de que foi feito um $Wait$, a thread que processa essa ligação vai ficar bloqueada na variável de condição. Já a thread encarregue de alterar as regiões do clipboard local, quando modifica uma região, lança o sinal que liberta as threads bloqueadas para a região em questão (função $pthread\_cond\_broadcast()$), que efetuarão um $Paste$ regular, como já tinha sido visto na Figura 3.

5 Gestão de recursos e tratamento de erros

Para finalizar, neste capítulo vão ser justificadas algumas escolhas feitas na implementação do projeto, bem como a resposta do programa a possíveis erros.

Particularidades do programa:

- Na região crítica, no segmento que bloqueia os acessos de escrita para a modificação de uma região no clipboard local ($send\_region()$ - regions.c), podia ter-se optado por copiar a mensagem da região em questão para uma *bytestream*, e fora da região crítica fazer o envio da mensagem. No entanto, foi decidido implementar o envio da mensagem diretamente de dentro da região crítica, pois apesar

da perda na velocidade de execução, a garantia de que a mensagem é enviada é superior (a alocação de memória para a *bytestream* pode falhar). Optou-se por privilegiar uma maior garantia de um $Paste/Wait$ sucedido face a uma maior velocidade de execução que diminua esta probabilidade.

- Quando a lista de clipboards clientes é corrida, para realizar um $Copy$ no sentido descendente ($update\_region()$ - regions.c), por simplicidade de código, poderia ter-se chamado a função $Copy$ da API para enviar a mensagem em cada nó. Contudo, como a função seria chamada, por cada alteração numa região, uma vez para cada clipboard cliente ligado ao clipboard local, optou-se por escrever as linhas de código que realizam o envio da informação da mensagem seguido do envio da mesma, com o intuito de aumentar a velocidade de execução desta rotina.

- Após vários testes ao projeto com envio de mensagens de tamanho já significativo (de ordem igual ou superior a 1 MB), verificou-se que o trânsito de mensagens com uma única chamada da função $read()/write()$ não era suficiente para que a mensagem fosse transferida na sua totalidade. Para permitir a realização destes pedidos, foi então implementado um ciclo de escrita/leitura, que chama as funções referidas até que a mensagem ordenada seja integralmente enviada/recebida. Por outro lado, é sabido que esta implementação pode gerar alguns problemas no tratamento de erros, como por exemplo levar a que uma thread fique presa num ciclo de leitura. Para não descartar nenhuma das hipóteses foi definida a variável $MESSAGE\_FLUX\_CYCLE$ no ficheiro *clipboard_imp.h* que estipula a implementação com o ciclo de escrita/leitura, pelo que basta apagar/comentar a linha que define esta variável para que seja estipulada a outra implementação.

Resposta do programa a possíveis erros:

- Quando no decorrer do programa é executado o comando **Ctrl^C**, é lançado o sinal $SIGINT$ que, neste projeto, foi programado para fechar o ficheiro utilizado para as comunicações locais (UNIX), e depois então terminar o programa com $exit(0)$.

- Todas as *system calls* que falhem na execução do programa, estão protegidas para que nesta ocorrência, seja escrito no terminal onde sucedeu o erro, seja fechado o ficheiro utilizado para as comunicações locais, e seja terminado o programa com $exit(-1)$. Chama-se à atenção de que foram exceptuadas desta proteção as funções $read()$ e $write()$, que quando falham apenas se fecha ligação com a qual houve a falha, e é terminada a thread correspondente. É ainda oportuno referir que quando falha um $write()$ é lançado o sinal $SIGPIPE$ que, por defeito, termina o programa. Para impedir este evento, é chamada uma função que ignora este sinal assim que é lançado o clipboard local.

- Argumentos de entrada inválidos, isto é, que não cumpram o formato definido para o lançamento de um clipboard em modo *single/connected*, causam o seu término imediato com $exit(-2)$.

- Quando no sentido ascendente de um $Copy$ falha a alocação de memória que guardaria a mensagem, a mensagem é simplesmente perdida. Já no sentido des-

cendente, visto que há a possibilidade da alteração já ter sido feita em outros clipboards da rede, a mensagem não pode ser ignorada pois seria perdido o sincronismo. Esta falha é então processada como qualquer outra falha na execução de uma *system call*.

- Quando um programa ligado ao clipboard local é fechado, no próximo trânsito de informação que com ele seria feito, vai haver essa deteção e é fechado também, no clipboard local, o *endpoint* para a ligação. No caso da ligação ser o clipboard servidor, é de seguida implementada a simulação já referida para o modo *single*, sendo que desde o fecho do ponto de ligação até à conclusão da simulação de um clipboard servidor, são bloqueadas as comunicações no sentido ascendente. Caso contrário, é terminada a thread que geria a ligação e, na ocorrência de ser um clipboard cliente, será posteriormente removido o respetivo nó da lista ligada de clipboards clientes.

- Na eventualidade de ser pedido por um cliente um $Copy$ onde o tamanho especificado seja diferente do tamanho real da mensagem, não haverá nenhuma repercussão no clipboard local, pois com o protocolo utilizado é sabido o tamanho (em bytes) pressuposto de cada transição, pelo que sempre que a informação transferida difere, em tamanho, do que se espera, a operação é abortada.

- Supondo, para finalizar, que uma aplicação maliciosa efetua duas operações, em simultâneo e pelo mesmo ponto de ligação. Vai suceder-se um conflito de mensagens pelas duas operações, que é desconhecido no clipboard local. Este vai então dessincronizar o protocolo de comunicação com esta ligação, pelo que vai fechá-la e terminar a thread que geria a ligação.