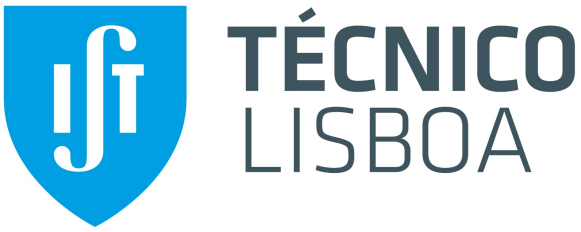


Programação de Sistemas

MEEC

Relatório do Projeto

Grupo Nº 23

Autores:

Diogo Rodrigues (Nº 84030)
Leandro Almeida (Nº 84112)

11/06/2018

Conteúdo

1 Introdução

Este trabalho tem como objetivo a implementação de um clipboard distribuído, em que existem aplicações que podem fazer copy para o clipboard distribuído, paste e wait do mesmo, através de uma API predefinida.

É de se referir que o clipboard distribuído corre em diferentes máquinas, pelo que um copy numa determinada região num clipboard terá que estar disponível para um paste ou wait num outro clipboard, se os correspondentes clipboards estiverem ligados. Caso os clipboards não estejam conectados (*Single Mode*) a informação a partir desse mesmo atualizar-se-à apenas para os que estejam conectados a este.

Várias aplicações podem estar localmente ligadas ao mesmo clipboard pelo que se tem de ter em atenção alguns casos específicos como o de duas aplicações diferentes fazerem copy e paste ao mesmo tempo.

Assim, o sistema desenvolvido armazena qualquer tipo de informação num servidor. Este sistema é constituído por um servidor e uma API para a comunicação entre cliente e servidor. Cada clipboard pode comunicar ao mesmo tempo com varios clientes e com clipboards que estejam ligados à sua rede, pelo que o projeto apresenta uma arquitetura multi-threading.

A arquitetura do clipboard distribuído encontra-se representada na figura 1:

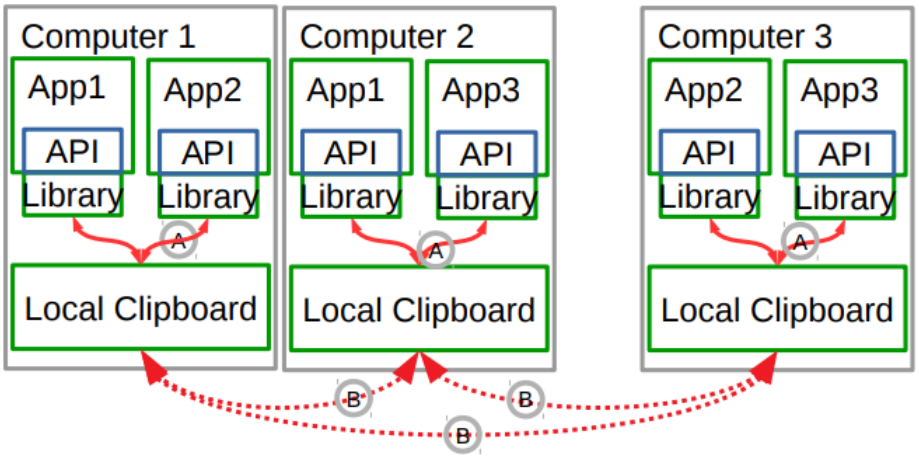


Figura 1: Arquitetura do programa clipboard distribuído(retirada do enunciado)

2 Estruturas de Dados

Para que seja possível fazer o envio das mensagens que são trocadas entre a aplicação e o clipboard local e entre os próprios clipboards é necessário definir as diferentes mensagens para realizar as funcionalidades pedidas. Desta forma, criou-se uma estrutura que contém a informação relevante para o emissor da mensagem naquele momento poder preparar o recetor dessa mesma mensagem para a informação que vai receber. Esta estrutura representa-se na figura 2.

```
typedef struct _message_struct  {
    int action;
    int region;
    size_t length;
} _message;
```

Figura 2: Estrutura enviada na comunicação para posteriormente se enviar o conteúdo da mensagem

- action: contém a ação a realizar, ou seja, se irá ser um copy, paste ou wait.
- region: a região que vai ser afetada por essa mensagem (inteiro entre 0 e 9).
- length: representa o número de bytes a serem enviados.

Assim, ao longo de todo o programa antes de ser enviada os dados propriamente ditos, primeiramente, envia-se uma estrutura com a informação da mensagem de forma a que o recetor (quer seja app ou clipboard) possa saber o que irá esperar e o que deve fazer com esses dados, que se irá ver em seguida.

A estrutura de dados de cada clipboard que contém a informação relativa a esse mesmo clipboard está representado na figura 3.

```
typedef struct clipboard_struct
{
    size_t size[MAX_REGIONS];
    void * matrix[MAX_REGIONS];
} _clipboard;
```

Figura 3: Estrutura que contém a informação do clipboard

- *matrix*: vetor de ponteiros, em que cada posição aponta para a informação contida nessa região nesse dado momento
- *size*: é um vetor com o tamanho (número de bytes) da informação contida em cada região.

De forma a saber quais os clipboards conectados com um dado clipboard remoto utiliza-se uma lista ligada. Esta lista contém o file descriptor do socket que comunica com o clipboard filho correspondente. Assim sendo, cada clipboard, no momento em que outro clipboard se liga a este, cria um novo nó na lista ligada onde *sock_fd* é o file descriptor do socket desta comunicação.

```
typedef struct _clipboards_list_struct
{
    int sock_fd;
    struct _clipboards_list_struct  * next;
} _clipboards_list;
```

Figura 4: Lista de filhos de um clipboard

As variáveis globais utilizadas encontram-se representadas na figura 5.

- main_server: ponto de ligação para o clipboard servidor (pai).

- mux_sendUP: utilizado para manter a sincronização quando há propagação de informação entre clipboards no sentido ascendente.
- rwlock: usado para copy/paste que são efetuados ao mesmo tempo - *race conditions*. Bloqueia os acessos à *matrix* bem como ao *size* do clipboard. Na função paste faz-se readlock para que várias threads possam ler ao mesmo tempo. No copy writelock para que só uma escreva.
- mutex_child: usado para prevenir *race conditions*, quando se mexe na lista de filhos do clipboard e quando se inicializa o filho.
- mutex_wait: como o nome indica, é usado no wait para implementar a variável de condição.
- data_cond: utiliza-se no wait. Permite a espera inativa de uma determinada região até que esta seja atualizada.

```
int sock_main_server;

pthread_mutex_t mux_sendUP;
pthread_rwlock_t rwlock[MAX_REGIONS];
pthread_mutex_t mutex_child;
pthread_mutex_t mutex_wait[MAX_REGIONS];
pthread_cond_t data_cond[MAX_REGIONS] = PTHREAD_COND_INITIALIZER;
```

Figura 5: Variáveis globais usadas para sincronizar e condições de corrida

3 Arquitetura do Sistema

O sistema é composto essencialmente por 3 componentes, a API responsável por ser a interface que permites programadores desenvolverem aplicações que usem o clipboard distribuído, a biblioteca que implementa a API com o código e funções para a aplicação interagir com o clipboard e, ainda, o clipboard local que é o componente mais importante e que é responsável por receber conexões tanto das aplicações locais como de outros clipboards que possam estar a correr noutros computadores (clipboards remotos). Este componente recebe os vários pedidos das aplicações, guarda os dados e replica-os para toda a rede de clipboards remotos.

O clipboard pode começar em *single mode* (onde o clipboard não se liga a nenhum outro, no entanto existe a possibilidade de outros se ligarem a ele) ou conectado a outro. Caso o clipboard arranque no modo conectado, ligando-se a outro já existente, passa a fazer parte desta rede pelo que uma alteração em qualquer um replicar-se-à por toda a rede. Esta replicação de informação é feita de forma a que se preserve o sincronismo. Cada clipboard possui 10 regiões diferentes, todas acedidas independentemente, onde é possível para o utilizador enviar mensagens para uma região pretendida, se existente.

A figura 6 mostra a função *main* presente no ficheiro *clipboard.c*.

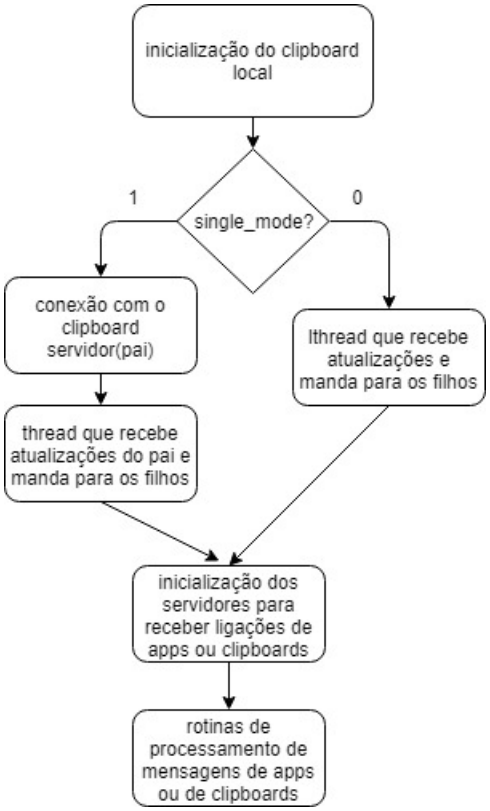


Figura 6: clipboard.c

Para começar, na inicialização são criadas todas as variáveis a usar ao longo de todo o programa. Se o clipboard estiver em modo *single* as suas regiões são inicializadas a NULL. Se estiver conectado a um outro clipboard, é logo lançada uma thread que recebe todas as atualizações do clipboard servidor, e é logo inicializado com todo o seu conteúdo. São lançadas as rotinas que recebem ligações de clientes e, paralelamente, a de ligação de novos clipboards (nova thread). Assim sendo, são estas que irão ser responsáveis por todo o desenrolar do processo.

3.1 Protocolo de Comunicação

Como já se abordou, existem dois tipos de ligação estabelecidos no programa: entre clipboard-aplicação e entre clipboards. Para estes dois tipos de comunicação são utilizados sokets do tipo *stream*. Para a comunicação local com o cliente são utilizados o domínio UNIX e para os outros dois tipos de comunicação são utilizados domínio INET. O protocolo estipulado para o tipo UNIX passa por receber uma estrutura que contém a região pretendida (esta tem de estar entre 0-9 senão viola-se o protocolo), e a ação pretendida (COPY, PASTE e WAIT, senão viola-se o protocolo), e consoante a ação pretendida o protocolo estabelecido trata de fazer a comunicação em sentidos diferentes. Isto é, uma vez que os dois tipos de comunicação possíveis são o envio de uma estrutura que permite depois que a mensagem seja enviada e após isso, o envio de um array que representa a própria mensagem, esta transmissão pode ser realizada (a título de exemplo para o caso UNIX) no sentido clipboard -> cliente ou cliente->clipboard consoante a ação pretendida.

Refere-se também que se usou sockets do tipo *stream* e não *datagram* de forma a garantir

que a informação é processada de forma ordenada e que não há perda de informção nas transmissões. O mesmo se aplica na escolha deste tipo de comunicação para o domínio UNIX, em relação à comunicação por um FIFO ou PIPE.

3.2 Fluxo de Tratamento de Pedidos

De forma genérica o aspeto das ligações entre as aplicações locais e o clipboard local respetivo, é a representada na figura 7:

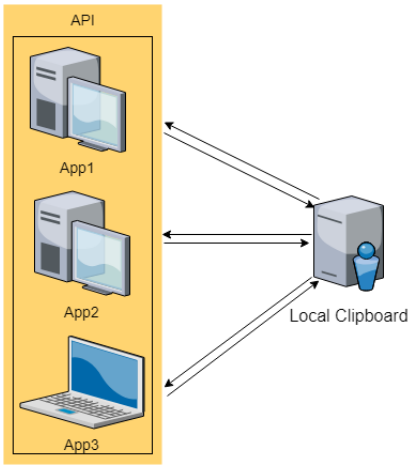


Figura 7: Ligações entre as aplicações locais e o clipboard local

A comunicação aplicação com o clipboard é feita através de comunicação unix, como já foi visto anteriormente. O cliente tem a opção de fazer copy, paste ou wait. No domínio do clipboard, uma thread está sempre à espera de receber novas conexões de clientes e, cada vez que algum cliente tenta estabelecer uma nova ligação, é criada uma nova thread para cada cliente de modo a ser possível responder ao mesmo tempo aos pedidos de todos os clientes, como representado na figura 8 para o caso em que estão 2 clientes ligados a um clipboard:

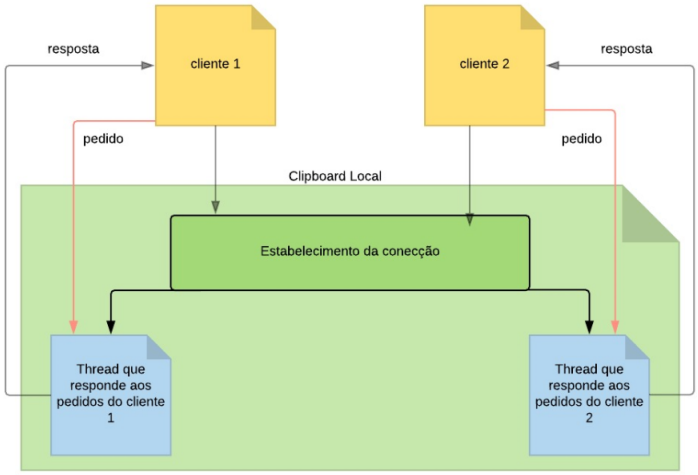


Figura 8: Threads do clipboard associadas as aplicações locais

O fluxo de tratamento de pedidos entre uma aplicação local e o clipboard está representado no fluxograma apresentado em baixo.

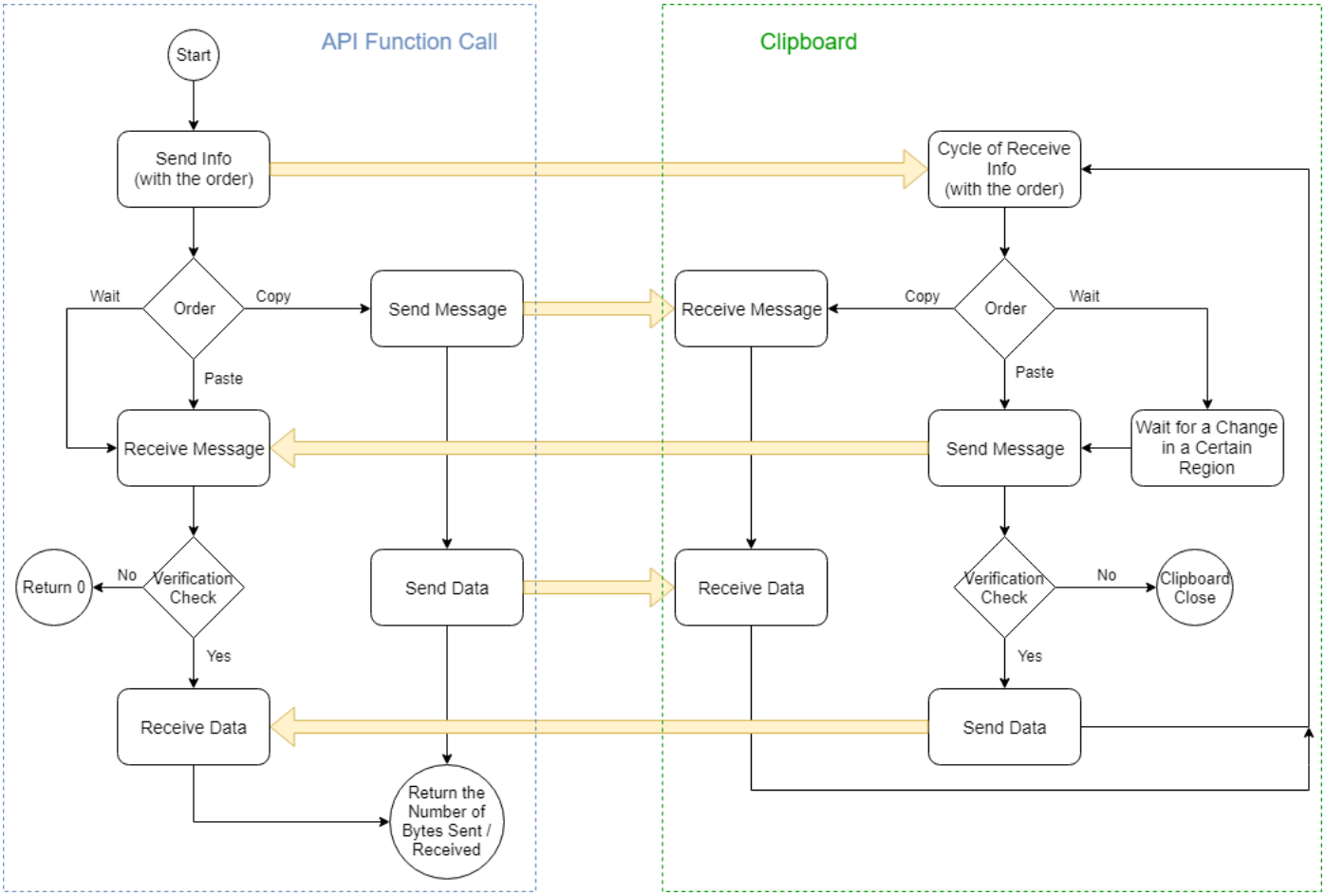


Figura 9: Fluxograma do fluxo de tratamento de pedidos

Como já visto, a primeira comunicação consiste em enviar uma estrutura de modo a se saber a ação pretendida bem como o tamanho da mensagem a enviar. É de se referir que, antes de tudo,a primeira coisa de todas a ser feita é verificar se os dados da mensagem são válidos. Isto é, verificar se a região e a ação enviadas são parâmetros válidos, pois se por alguma razão o cliente conseguir aceder ao código da library.c, ficando assim a conhecer o protocolo, existe a possibilidade de fazer uma biblioteca maliciosa que possa mandar o programa a baixo. Este procedimento é uma forma de prevenção.

De acordo com a ação pretendida tem-se que:

- Copy

O comando de copy é dado pelas aplicações locais ao clipboard local e consiste em enviar os dados que pretende que sejam guardados numa dada região. Quando o clipboard recebe uma instrução de COPY prepara-se para armazenar uma mensagem com o tamanho recebido na primeira comunicação e armazenar. Este processo inicia-se na thread responsável pela comunicação com aquela aplicação local que está à espera dos pedidos e que assim que recebe um novo pedido verifica qual a ação pretendida.

Quando se sucede um copy, os dados não são imediatamente guardados na região do clipboard, mas são sim enviadas para o topo da rede de clipboards (este processo vai ser visto ao promenor na secção seguinte) de modo a garantir a sincronização de todos os

clipboards da mesma rede. Isto pode não parecer a melhor solução à primeira vista, visto que a informação não é guardada imediatamente no clipboard, porém caso a informação fosse guardada logo no clipboard e só depois enviada para a restante rede traria problemas de perca da sincronização de toda a rede. Um exemplo simples deste problema está representada na figura 10:

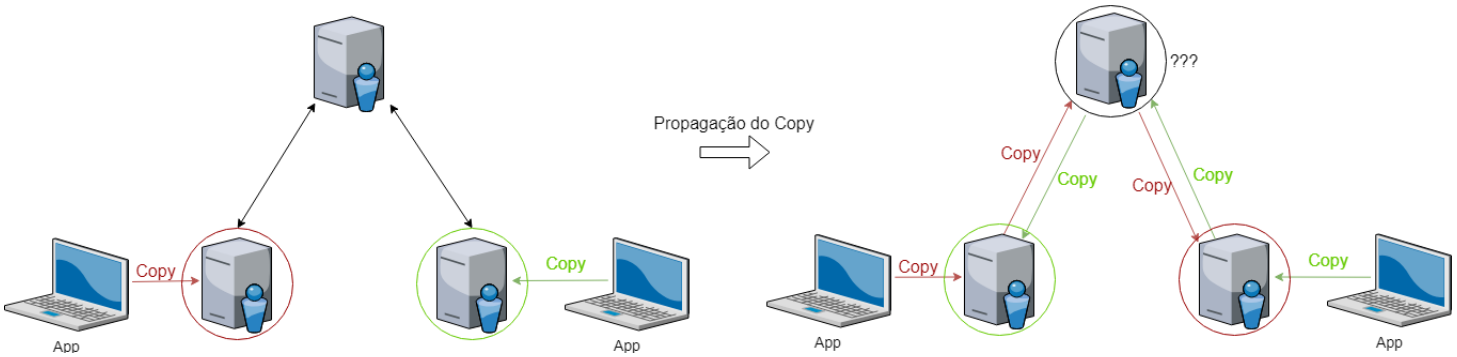


Figura 10: Exemplo de erro no copy

Neste exemplo, é possível ver o caso de duas aplicações realizarem um copy ao mesmo tempo em computadores diferentes (copy verde aplicação do lardo esquerdo e copy vermelho aplicação do lardo direito) e quando a informação é replicada para a restante rede, tem-se que a rede do lado esquerdo irá ficar com a informação referente ao copy verde e rede do lado direito irá ficar com a informação referente ao copy vermelho.

Para evitar este problema e manter sempre a sincronização, os dados só são guardados na região pretendida quando estão a descer na árvore e são enviados para todos os clipboards filhos, incluindo o que recebeu o copy. Todo este procedimento será explorado em seguida.

- Paste

Analogamente ao copy, o comando de paste é dado pelas aplicações locais ao clipboard local e consiste em copiar os dados de uma certa região do sistema para aplicação. Este processo tal como o copy inicia-se na thread responsável pela comunicação com aquela aplicação local e consoante o pedido de paste envia a informação pretedida. Se a região pretendida não contiver nenhum conteúdo ou se o tamanho que o utilizador coloca que está disposto a receber for inferior ao que a região efetivamente possui, é retornado 0 para a aplicação. Caso contrário é retornado o número de bytes enviados do clipboard para a aplicação.

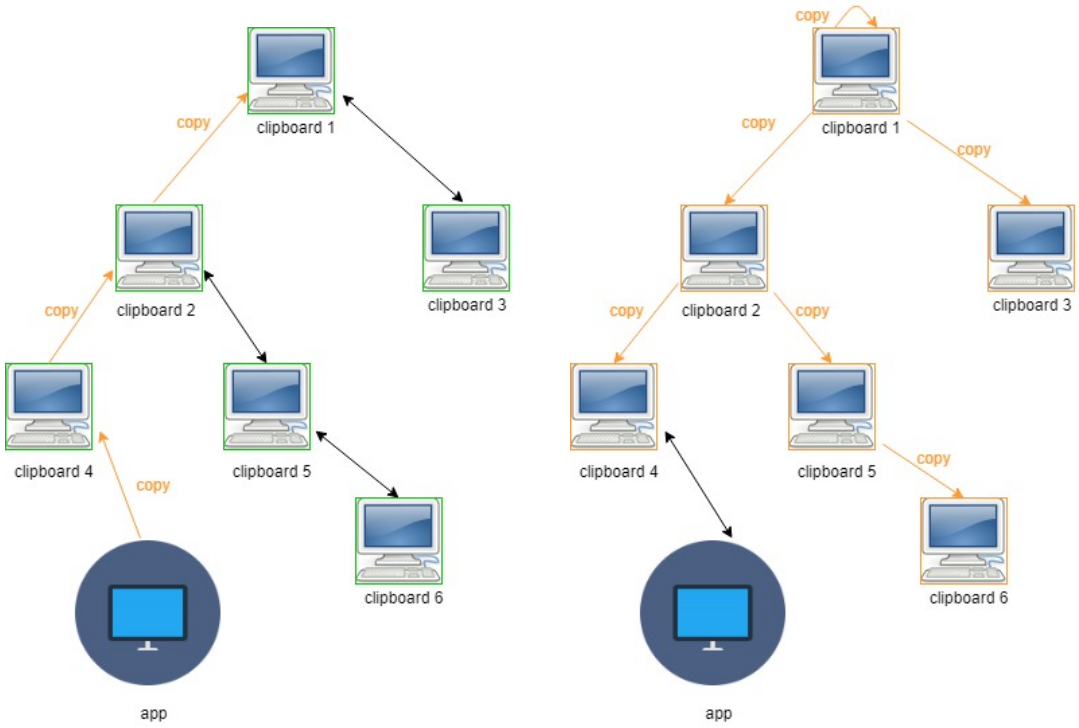
- Wait

A aplicação fica à espera de uma alteração numa dada região (novo copy) e quando tal acontecer, a função é exatamente igual à função *paste*, pelo que o procedimento é o mesmo.

3.3 Propagação de informação na rede e Gestão de threads

Quando se tem uma rede constituída por vários clipboards ligados uns aos outros, ao haver uma atualização num deles, todos os outros pertencentes à sua rede têm de sofrer a mesma atualização. A forma encontrada para garantir que toda a rede no final fica com a mesma informação (de forma a prevenir o caso já demonstrado na figura 10 é a seguinte:

Um clipboard quando recebe uma atualização, em vez de armazenar de imediato na sua *matrix*, envia a informação para cima até que chegue ao clipboard que criou a rede em modo *single* - propagação em sentido ascendente. Este clipboard armazena a informação e, posteriormente, envia-a para todos os seus filhos - propagação em sentido descendente. No sentido descendente os clipboards atualizam a *matrix* na região pretendida, enviando para os seus filhos até que se chegue à raíz da árvore. Este mecanismo encontra-se representado na figura 11.



(a) Transmissão no sentido ascendente (b) Transmissão no sentido descendente

Figura 11: Propagação da informação na rede quando há uma atualização num dos clipboards

Como se pode observar na primeira imagem, quando o clipboard 4 recebe um copy (cor de laranja) por parte da app, este não fica logo com a cor de laranja (permanece a verde que é o estado anterior), e envia a informação para o seu pai. Isto repete-se até que se chegue ao topo da árvore (sentido ascendente). Quando a informação chega ao topo da árvore, olhando para a segunda figura, percebe-se que o pai guarda a informação (passando por isso de verde para cor de laranja) e replica para todos os seus filhos. A replicação por todos faz com que no final fiquem todos sincronizados de cor de laranja (sentido descendente).

Para a propagação da informação da rede foram criadas threads como se pode observar na figura 12.

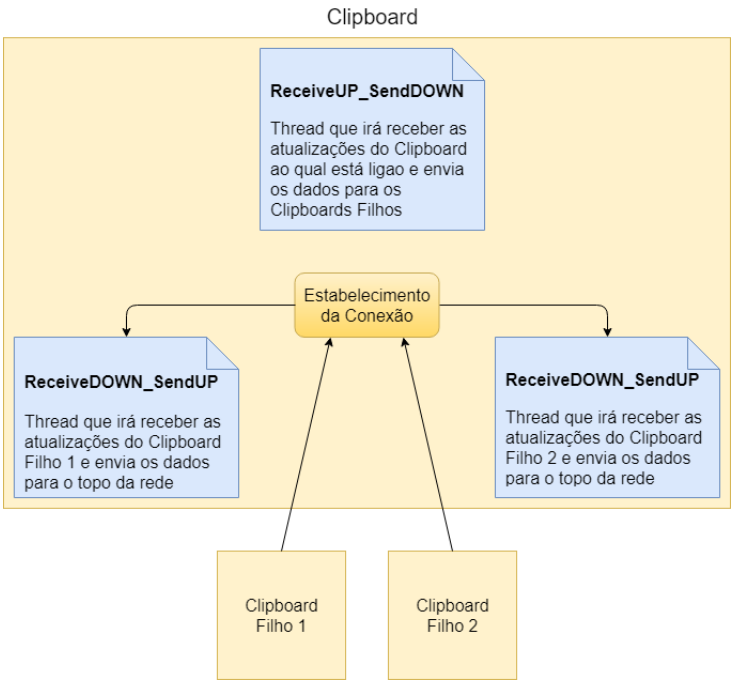


Figura 12: Threads presentes no funcionamento do clipboard

Para cada comunicação entre clipboards que recebe as atualizações vindas dos seus filhos/clipboard cliente, e envia para o seu pai/clipboard servidor – sentido ascendente na árvore, criou-se a thread *ReceiveDOWN_SendUP* que trata de reencaminhar a informação para cima sempre que é recebida de baixo.

Quando a informação atinge o topo da árvore - clipboard em modo *single*, a comunicação é feita utilizando as mesmas threads utilizadas pelos clipboards em modo conectado, no entanto a comunicação não se faz através de um socket, mas sim através de um pipe. Isto é, o clipboard que representa o topo da árvore recebe as atualizações através da thread utilizada no sentido ascendente (*receiveDOWN_sendUP*), e envia através de um PIPE para a thread que vai fazer a replicação para todos os seus filhos, através da informação recebida do seu 'pai' (que neste caso recebe através de si mesmo via PIPE). Deste modo, garante-se que não há perda de informação e que é processada uma mensagem de cada vez.

Chegando a informação ao topo da árvore, e tendo esta já aqui sido processada, a informação tem de ser replicada para todos os clipboards, no sentido descendente. Para isso, existe uma thread por cada clipboard responsável por receber as atualizações vindas do seu pai (ou PIPE) e reencaminhá-las para todos os seus filhos - *receiveUP_sendDOWN*.

Assim sendo, uma rede de clipboards teria o aspeto mostrado na figura 13, onde se apresentam mutex que irão ser abordados na seguinte secção.

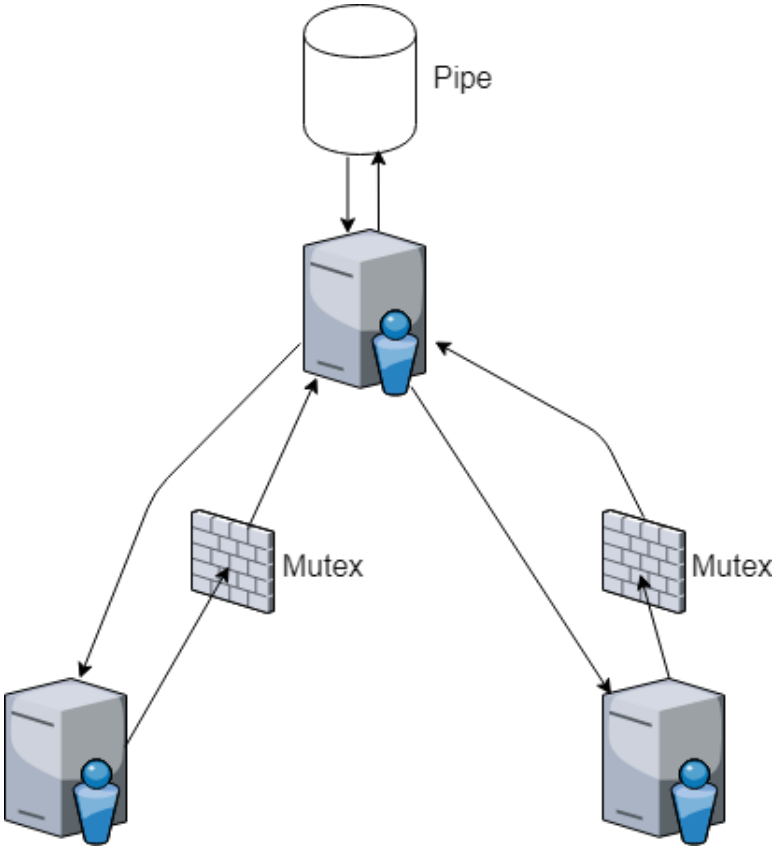


Figura 13: Rede de Clipboards

4 Sincronização

A arquitetura do sistema ter sido desenvolvimento com multi-threading traz muitos benefícios em termos de eficiência e rapidez devido ao paralelismo que consegue no fluxo de tratamento de pedidos, porém levanta alguns problemas ao nível de sincronimo. Ou seja, ao se ter várias threads a partilharem a mesma memória e com possibilidade de lhe aceder ao mesmo tempo, pode haver conflitos. Sendo que nesta secção vai-se perceber como é que esses conflitos foram evitados.

4.1 Identificação da Região Crítica

As regiões do clipboard local podem ser acedidas por várias threads, visto que se tem uma thread por cada aplicação local que existir. Desta forma, há que prevenir o caso de existir um copy e um paste/wait ao mesmo tempo, pois esta situação poderia gerar um conflito visto que um comando passa por alterar uma região e os outros por ler uma região, sendo que , no processo de atualizar a região, primeiramente a mensagem é eliminada e depois atualizada podendo entre essas duas situações haver uma leitura inválida da região.

Para resolver este caso recorreu-se à variável *rwlock* que no caso em que esteja a ser escrito na memória bloqueia todos os acessos e que no caso em que esteja a ser lido da memória bloqueia os acessos de escrita na memória. Desta forma, quando estiver a ser feito um copy garante-se que nenhum paste/wait está a ter acesso a fazer leituras da memória e vice-versa. Esta variável, já apresentada anteriormente, é um vetor com 10 posições, pois

podem estar a haver copy's e paste's/wait's ao mesmo tempo desde que sejam de regiões diferentes.

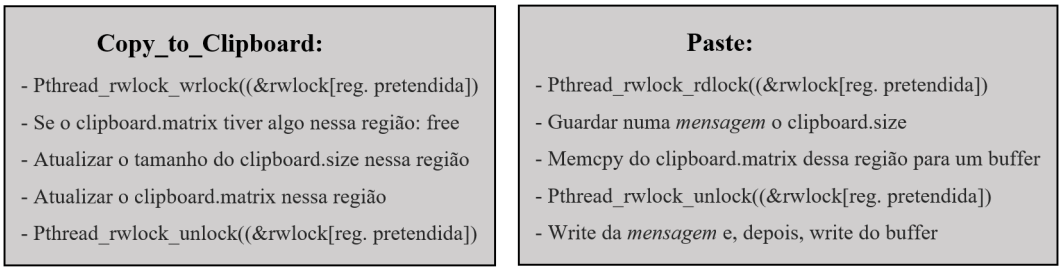


Figura 14: Região Crítica

Enfatiza-se que, de modo a encurtar a região crítica, na função Paste foi feito um memcpy do que está no *clipboard.matrix* para um buffer dentro da região crítica possibilitando assim que o write do conteúdo da região seja só feito depois de se dar unlock da região crítica. Desta forma, os *write*'s da estrutura da mensagem, bem como do conteúdo são dados fora da região crítica (enviando assim a variável local), o que faz com que se evitem duas idas ao kernel, pelo facto de o *write* ser uma *system-call*.

4.2 Implementação de Exclusão Mútua

Na comunicação no sentido ascendente, tem de se ter em conta que se pode ter várias threads a escrever para o mesmo socket, pelo que tem de se utilizar exclusão mútua através de um mutex (ver mutex na figura 13), de modo a que se escreva uma de cada vez, mantendo-se assim o sincronismo. Se isto não acontecesse, um clipboard que estivesse a receber atualizações de diferentes filhos poderia receber de um deles a estrutura com a indicação da região e do tamanho da mensagem e, em seguida, apesar de estar à espera de receber a mensagem desse mesmo filho, recebe primeiro informação vinda de um outro filho, perdendo-se o sincronismo. Assim na thread responsável de receber as informações de um filho e enviar para o pai, bem como na função que envia para o pai uma atualização vinda do cliente a zona critica que garante a sincronização é dada por:

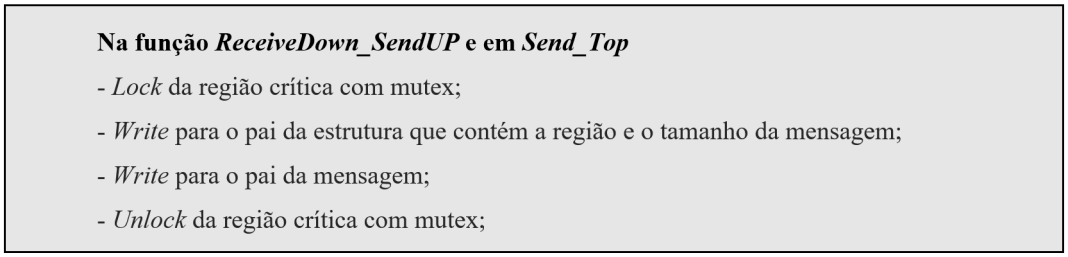


Figura 15: Exclusão mútua no sentido ascendente

Para garantir um caso de sincronização e, ao mesmo tempo, de condição de corrida à lista que contém os *sock_fd* dos filhos, foi criado um mutex utilizado para que haja exclusão mútua em funções como *create_new_son* e *delete_son*, bem como sincronismo a enviar as atualizações para os filhos (garantir por exemplo que quando se está a inicializar um filho - após este se ligar ao pai – não há outra atualização proveniente do cliente ou de outro clipboard a ser enviada ao mesmo tempo).

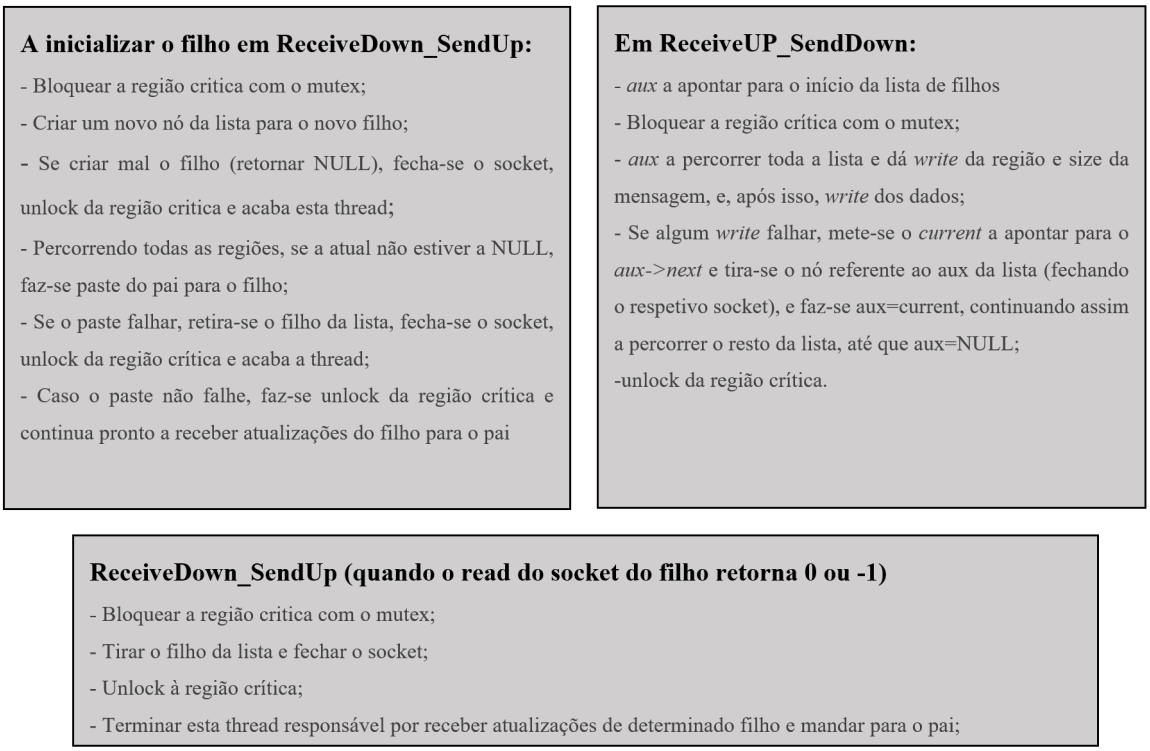


Figura 16: Exclusão mútua no sentido descendente aquando de manipulação da lista de filhos

Esta situação foi verificada introduzindo um getchar() durante a inicialização de um filho quando este se liga ao pai, para que fique parado dentro da região crítica. Antes de pressionar qualquer tecla fez-se um copy, para ver se o sincronismo era mantido. O pai só atualiza o filho deste copy após a inicialização ser concluída.

4.3 Implementação da espera não ativa para o Wait da API

Para que o comando o wait possa ser realizado, são necessários mecanismos adicionais para esperar dentro de um região critíca que são as chamadas variáveis de condição. Estas variáveis permitem que seja possível várias threads estejam a "dormir"dentro de uma região crítica.

Posto isto, tal como definido na secção das Estruturas de dasdos, foi criado um vetor de mutex's assim como um vetores de variáveis de condição de modo a que as 10 regiões possam funcionar de forma independe umas das outras, tal como se mostra na figura .

```
if(pthread_mutex_lock(&mutex_wait[message.region])!=0)
    error_confirmation("Error in lock mutex_wait in function wait");

//waiting for the new signal
if(pthread_cond_wait(&data_cond[message.region], &mutex_wait[message.region])!=0)
    error_confirmation("Error in cond_wait of mutex_wait in function wait");

//go out of the critical region and do paste
if(pthread_mutex_unlock(&mutex_wait[message.region])!=0)
    error_confirmation("Error in unlock mutex_wait in function wait");
```

Figura 17: Parte do código que demonstra a espera não ativa aquando da execuçao do comando de wait

Na figura acima pode-se ver que caso exista vários wait's numa determinada região, as threads correspondentes vão ficar bloqueadas nesse wait até que um novo copy seja feito e quando esse copy for feito no momento em que for colocado os novos dados no clipboard (sentido descendente da propagação na rede) irá ser lançado um sinal que "acorda"todas as threads que estejam adormecidos (figura 18).

```
if(pthread_mutex_lock(&mutex_wait[message_aux.region])!=0)
    error_confirmation("Error mutex_wait: function receiveUP_sendDown");

if(pthread_cond_broadcast(&data_cond[message_aux.region])!=0)
    error_confirmation("Error in broadcast in function receiveUp_sendDown");

if(pthread_mutex_unlock(&mutex_wait[message_aux.region])!=0)
    error_confirmation("Error in unlock mutex wait in function receiveUp_sendDown");
```

Figura 18: Parte do código onde é lançado o sinal que desbloqueia todas as threads em espera

5 Gestão de Recurso e Tratamento de Erros

Primeiro que tudo, há que referir que todas as funções do system-call quando invocadas tiveram tratamento de erro. Para funções como por exemplo pthread_create, pthread_mutex_lock se não retornarem sucesso, o clipboard termina, como exemplificado na figura 19:

```
if(pthread_create(&thread_id[2], NULL, user_communication, &user_fd)!=0)
    error_confirmation("Error creating thread user_communication:");
if(pthread_detach(thread_id[2])!=0)
    error_confirmation("Error in pthread_detach- receiveUP_sendDown:");
```

(a) Exemplo de tratamento de erros na criação de threads

```
if(pthread_mutex_lock(&mutex_child)!=0)
    error_confirmation("Error in lock mutex child in function receiveDown_sendUp");
```

(b) Exemplo de tratamento de erros nos mutex criação de threads

Figura 19: Exemplo de tratamento de erros

Para o caso de falhas em read e write a análise feita é mais cuidadosa. Se um write ou um read entre o clipboard e a aplicação falhar, fecha-se o socket que permite a comunicação entre essa mesma app e o clipboard.

No caso de ocorrer uma falha num read ou write entre clipboards essa ligação tem de ser fechada, nomeadamente no caso da comunicação falhar com um filho, para além de fechar o socket correspondente a essa ligação esse mesmo filho também terá que ser retirado da lista de filhos de forma a não receber outra qualquer comunicação. No caso da comunicação falhar com o pai, é desencadeado um processo que torna esse clipboard no servidor principal, fazendo desta forma que aquando da propagação na rede no sentido ascedente ele deixe de escrever no socket para o pai e passe a escrever para o PIPE.

É de se referir também que quando o clipboard recebe uma mensagem da aplicação a primeira coisa de todas a ser feita é verificar se os dados da mensagem são válidos. Isto é, verificar se a região e a ação enviadas são parâmetros válidos, pois se por alguma razão o cliente conseguir aceder ao código da library.c, ficando assim a conhecer o protocolo, existe a possibilidade de fazer uma biblioteca maliciosa que possa mandar o programa a baixo.

Um dos testes feitos foi ligar duas app's ao mesmo clipboard e uma delas fazer wait de uma determinada região (ficando assim uma thread do clipboard 'presa' no pthread_cond_wait à espera de um sinal de que a região foi modificada). De seguida essa mesma app faz CTRL+C, saindo do programa. Após isto, a outra app faz copy para essa mesma região. Uma vez que ao fazer copy, a função copy_to_clipboard faz um pthread_cond_broadcast, dando indicação para a outra thread que estava à espera do sinal para fazer um write num socket que já está fechado. Para que um write num socket já fechado (SIGPIPE) não mande todo o programa a baixo (ignorando apenas), introduz-se a seguinte linha de código tanto no lado do clipboard como na library:

```
void (*close_socket)(int);
close_socket = signal(SIGPIPE, SIG_IGN);
```

Figura 20: Evitar o SIGPIPE

No caso de erros de alocação de memória, nomeadamente nos buffers para guardar os dados na região em causa, o programa é encerrado visto que ou se esgotou a memória ou aconteceu algum problema naquele clipboard não havendo condições para que o mesmo possa continuar a correr.

Outro ponto que foi tido em conta foi na geração do porto ao qual outros clipboards poderiam-se conectar.

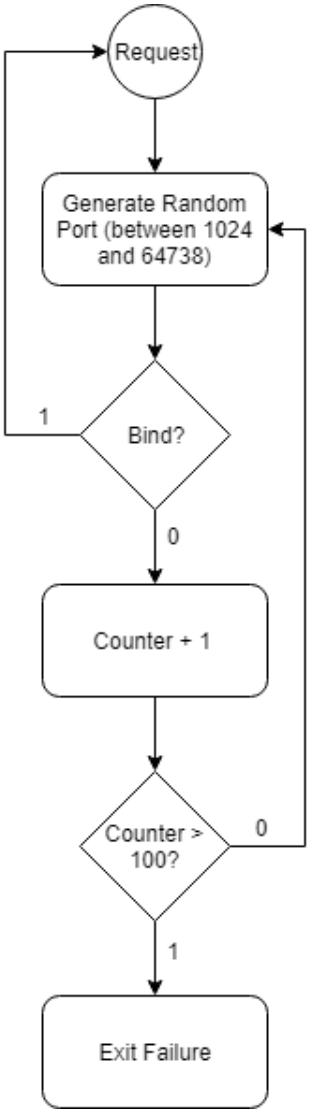


Figura 21: Gerar porto aleatório e fazer bind do mesmo

Desta forma, sempre que o porto é gerado e posteriormente é feito bind, se este porto já tiver sido atribuído vai dar erro, sendo que é gerado outro porto aleatório e voltasse a tentar atribui-lo. Este processo é realizado 100 vezes (100 foi apenas o número de referência utilizado).