Instituto Superior Técnico

Programação de Sistemas
MEEC 2017-2018

Área de transferência distribuída

Grupo nº 6

Alunos:
Carlos David Coito Pires, 84021
Rúben dos Santos Gomes, 84180

Docente:
João Nuno Silva
Ricardo Miguel Martins

Conteúdo

1 Introdução e objectivos de implementação

Este projeto consiste no desenvolvimento de uma área de transferência de dados num sistema distribuído de servidores, onde os dados são partilhados por todos os servidores. Os dois constituintes principais do processo são os servidores de armazenamento de dados em dez regiões distintas e os clientes que têm a possibilidade de interagir com o sistema distribuído.

Os clientes têm a possibilidade de realizar três ações através de uma API: copiar dados para uma das regiões de armazenamento, colar dados de uma das regiões, e esperar que uma das regiões seja alterada para posterior receção dos dados introduzidos. O servidor armazena todos os dados introduzidos pelos clientes e responde aos seus pedidos. Estes servidores têm requisitos de sincronização pois pretende-se que todos partilhem os mesmos dados e que possam responder a vários pedidos de clientes em simultâneo.

A arquitetura deste projeto é multi-threaded pois cada servidor pode estabelecer conexões e receber dados de vários clientes e de vários serviores, tendo por isso que ter várias threads a gerir as ações.

As diferentes conexões entre os servidores pode ser pensada como uma árvore. Se um dos servidores se desconectar poderá ocorrer a existência de mais do que uma árvore independentes capazes de responder aos pedidos de clientes, tal como anteriormente.

2 Estruturas de dados utilizadas

Para permitir o armazenamento da informação no servidor, utilizou-se um vetor com 10 posições. Cada posição apresenta dois campos distintos, uma região que guarda a informação (do tipo void *) e outra que contém o tamanho dos dados. Em todos os servidores lançados, esta estrutura de dados é inicializada com todos os tamanhos a zero e os ponteiros para armazenar os dados apontam para NULL (posição do *Clipboard* vazia).
Visto que cada *Clipboard* pode ter vários ligados a si, utilizou-se uma lista ligada que contém todos os *file descriptors* de servidores ligados a si. Esta estrutura, é usada para enviar a informação para os servidores "filhos"quando uma posição no *Clipboard* é atualizada, ou seja, esta lista contém a informação necessária para se construir a ligação entre servidor "pai"e "filho". Nesta estrutura, encontra-se também o identificador da *thread* responsável pela comunicação com o servidor "filho". O motivo pelo qual se escolheu o uso de uma lista, em vez de um vetor, foi o facto de haver necessidade de a percorrer ao enviar informação atualizada para os filhos, pelo que a propriedade de acesso rápido comum aos vetores não seria utilizada. Além disso, aquando da saída da rede de servidores por parte de um filho, a remoção desse elemento de uma lista é realizada de forma menos complexa do que a remoção de um vetor.
Visto que para um *Clipboard* é importante saber a quem é que este se ligou e o número de ligações que recebeu, criou-se uma estrutura que contém dois campos. Um para guardar o *file descriptor* do servidor "pai"e outro com o número de servidores que se ligaram a este. Se um servidor não se conectou a outro então a informação acerca do servidor "pai"fica a -1.

3 Arquitetura do sistema

A figura 1, demonstra as relações entre os clientes/aplicações e os diversos *Clipboards*. Um cliente apenas se conectar a um servidor que esteja a funcionar na mesma máquina. Cada *Clipboard* pode receber ligações de vários clientes e tratá-los ao mesmo tempo, situação representada pelo computador 3. As aplicações funcionam de forma independente e não estabelecem nenhuma ligação entre elas.

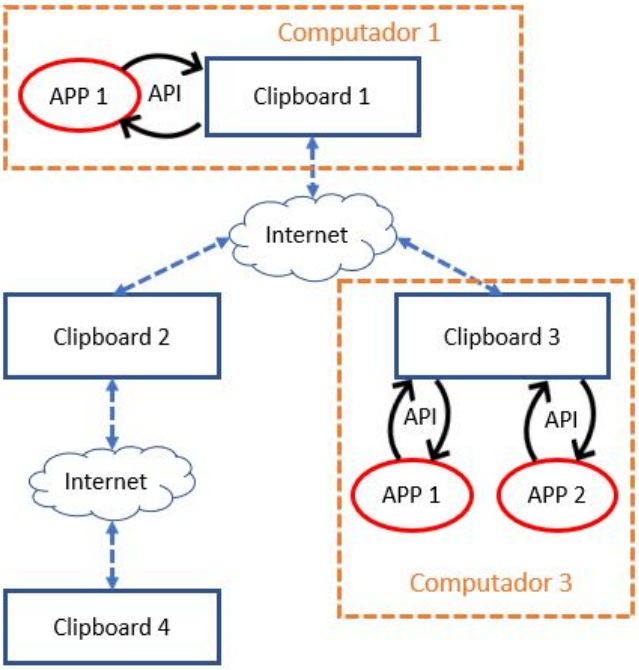

Figura 1: Exemplo de uma arquitetura global

Na figura anterior, os servidores encontram-se ligados uns aos outro. Quando um deles se desconecta a ligação com o seu "pai"e com todos os seus súbditos deixa de ser estabelecida. Na figura 2 encontra-se esquematizada a situação em que o *Clipboard 2* desconecta-se. Assim sendo, a ligação entre o *Clipboard 1* e *Clipboard 2* desaparece e consequentemente o *Clipboard 4* passa a estar em modo singular, visto que o caminho para o *Clipboard 1* foi eliminado.
Apesar da eliminação, o servidor "pai"e os seus súbditos não são afetados e continuará a ser possível conectarem-se novos servidores a eles.

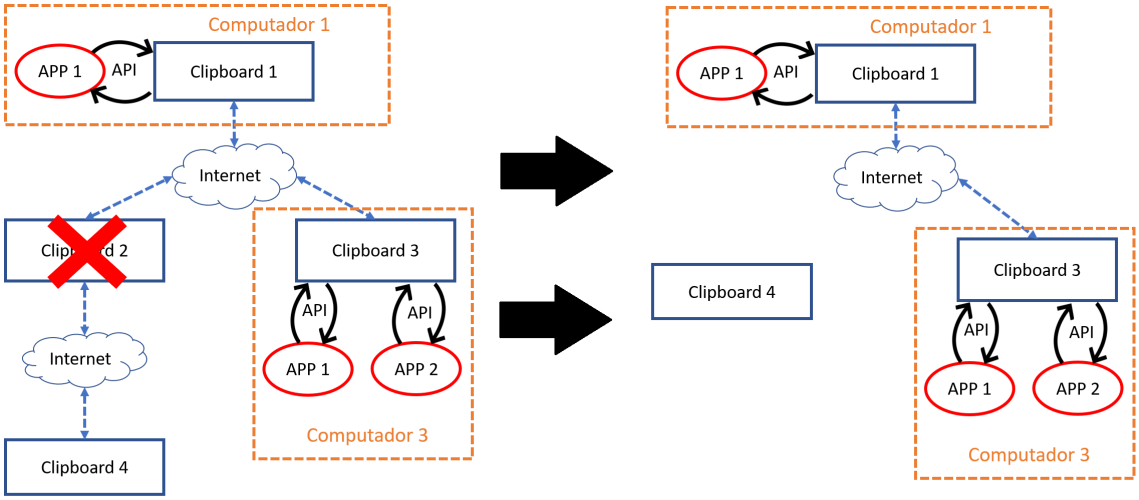

Figura 2: *Clipboard 2* desconecta-se, resultando em 2 árvores independentes

3.1 Registo de servidores

Inicialmente um servidor tem de ser lançado em modo singular. A partir desse momento, clientes podem estabelecer conexão com o servidor para realizar pedidos e servidores podem estabelecer conexão com o servidor para formar um sistema distribuído.

Quando um servidor estabelece conexão com um pré-existente (pai) descarrega em primeira instância todos os dados presentes até ao momento. De seguida cria uma thread que espera conexões de outros servidores (filhos) e que é responsável por transmitir os dados caso algum se conecte e uma thread que espera conexões de aplicações.

Cada servidor cria uma thread por cada aplicação que se conecte a ele que trata dos pedidos efetuados pelo cliente e cria duas threads por cada servidor que se conecte a ele: uma thread que recebe mensagens dos filhos e envia para o pai e uma thread que recebe mensagens do pai e envia para os filhos. Na figura 3 apresenta-se o fluxograma da thread principal clipboard.c.

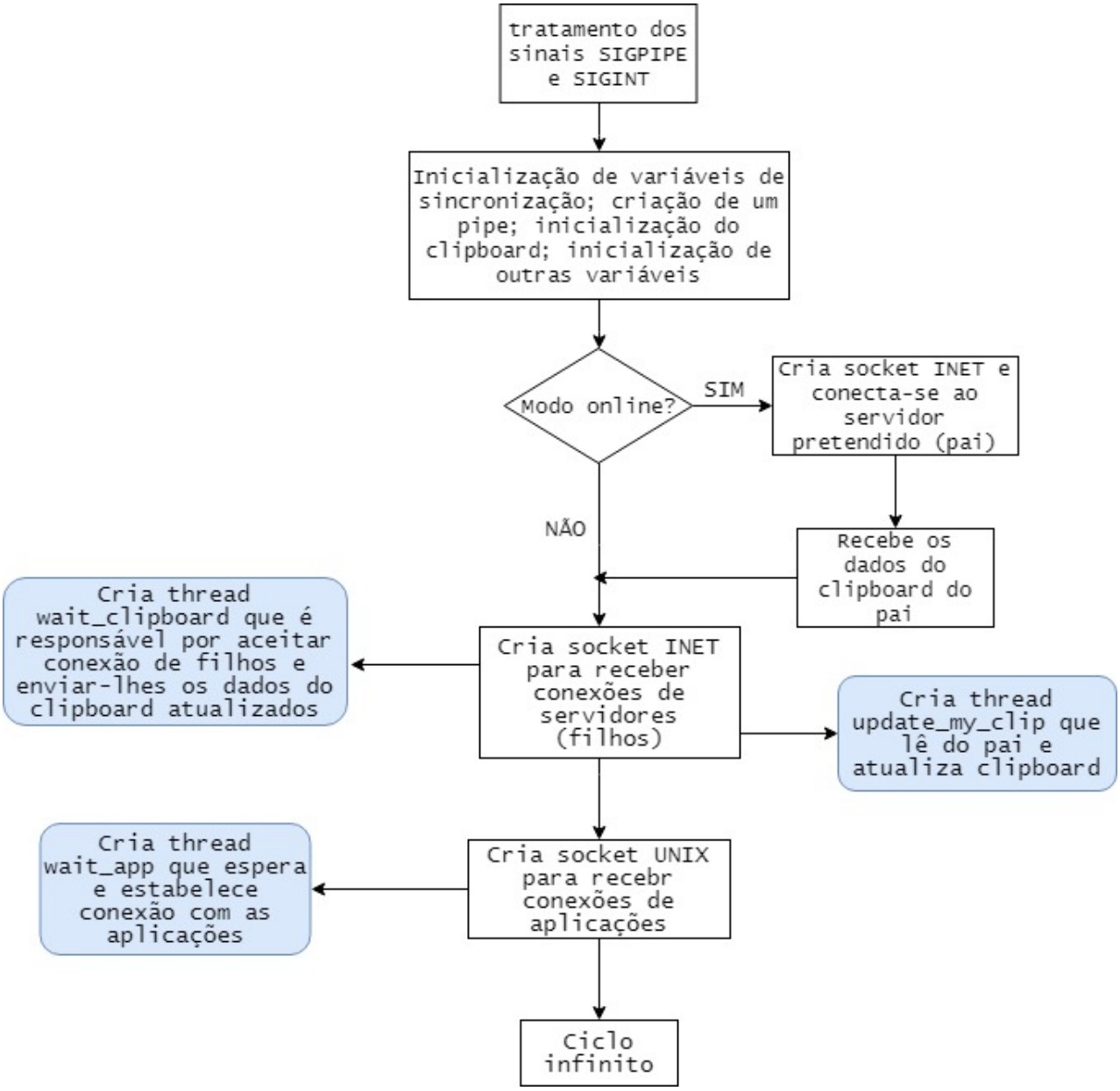

Figura 3: Fluxograma da thread principal clipboard.c

3.2 Protocolo de comunicação

Como já se viu anteriormente, para que o sistema global funcione é necessária a troca de informação a dois níveis. Primeiro dentro da mesma máquina, quando o cliente comunica com o servidor local, esta comunicação é feita através de *sockets* de domínio AF_UNIX. Para que a conexão entre servidor e cliente seja estabelecida o primeiro precisa de aceitar o pedido de ligação do segundo. Após o estabelecimento da ponte de comunicação é possível transmitir os dados nos dois sentidos. Implementaram-se sockets do tipo SOCK_STREAM pelo a comunicação é um fluxo contínuo de dados, a mensagem é fragmentada em pacotes, ele garante a entrega e a ordem dos pacotes.

Quando existem mais servidores em rede, é necessário que a informação esteja sincronizada, portanto é imperativo enviar os dados atualizados para os outros *Clipboards*.

A ligação entre *Cipboards* é feita através de *Sockets* do domínio AF_INET visto que, é preciso comunicar entre máquinas em locais diferentes. O *socket* responsável pela comunicação entre servidores é do tipo SOCK_STREAM, tal como a comunicação servidor e cliente. Portanto sabe-se que o protocolo de comunicação é TCP, conexão orientada.

3.3 Fluxo de tratamento de pedidos

3.3.1 Entre a aplicação e o servidor

Para que o servidor consiga responder ao pedido da aplicação é necessário haver mais do que uma troca de mensagens entre a aplicação e o servidor de forma a que este assimile qual a função a desempenhar e posteriormente enviar a resposta ao cliente. De seguida explica-se o fluxo de tratamento dos 3 pedidos:

Copiar

Nesta funcionalidade a comunicação é unidirecional, ou seja, as duas mensagens trocadas entre a aplicação e o servidor têm origem na aplicação e destino no servidor, uma vez que o cliente ao fazer "copy"não está à espera de receber nenhuma mensagem do servidor, mas apenas que este armazene os dados na região pretendida. A primeira mensagem enviada pela aplicação transporta informação sobre o tipo de ação pretendida (copiar), sobre o tamanho dos dados que vão ser enviados e sobre a região pretendida. A segunda mensagem enviada pela aplicação transporta os dados a serem armazenados na área de transferência. Assim, do lado do servidor após a primeira leitura é alocada a memória necessária para armazenar os dados e de seguida lê-se os dados enviados pela aplicação e armazena-se na área de transferência.

Colar

Nesta funcionalidade a comunicação é bidirecional, uma vez que após o pedido, o cliente está à espera de receber o conteúdo área de transferência da região pretendida. Primeiramente procede-se ao envio de uma mensagem para o servidor que transporta informação sobre o tipo de ação pretendida (colar), sobre o tamanho dos dados que o cliente pretende receber e sobre a região pretendida. Após a leitura da primeira mensagem o servidor envia uma mensagem com o tamanho dos dados que estão armazenados naquela região e logo de seguida, envia os dados. Do lado da aplicação, após uma primeira leitura aloca-se o espaço necessário para receber os dados e de seguida recebe-se os dados que estavam armazenados naquela região. Por fim, é feita uma cópia dos dados para um *buffer* com o tamanho requerido pelo cliente.

Esperar

Nesta funcionalidade o fluxo de tratamento de pedidos é muito semelhante ao fluxo descrito anteriormente na ação "colar". A comunicação é bidirecional pela mesma razão. Primeiramente procede-se ao envio de uma mensagem para o servidor que transporta informação sobre o tipo de ação pretendida (esperar), sobre o tamanho

dos dados que o cliente pretende receber e sobre a região pretendida. Após a leitura da primeira mensagem o servidor fica "adormecido"à espera de uma alteração na área de transferência nessa região. Quando houver alteração nessa região é lançado um sinal e os servidores que estão "adormecidos"à espera de alterações dessa região "acordam", e enviam uma mensagem com o tamanho dos dados que substituíram o conteúdo existente na região e logo de seguida, enviam os dados. Do lado da aplicação, após uma primeira leitura aloca-se o espaço necessário para receber os dados e de seguida recebe-se os dados que estavam armazenados naquela região. Por fim, é feita uma cópia dos dados para um *buffer* com o tamanho requerido pelo cliente.

3.3.2 Entre dois servidores

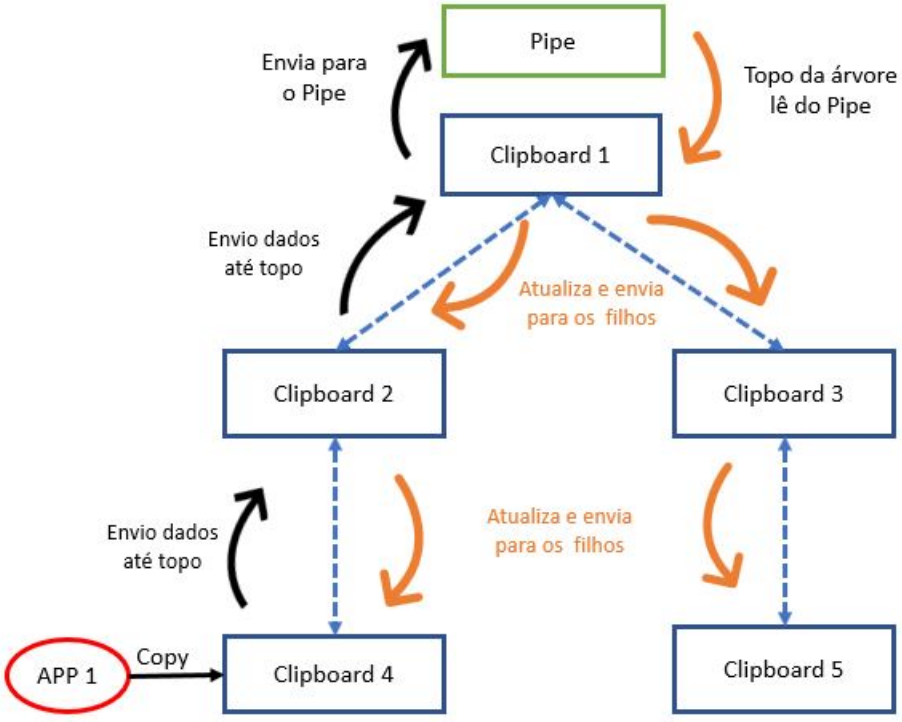

Figura 4: Exemplo de uma árvore de *Clipboards* e atualização da informação

É necessário que a mesma informação seja partilhada por todos os servidores em qualquer instante em que o servidor esteja pronto para responder a um pedido. Deste modo, ao receber uma atualização (num "copy") o sistema terá de replicar a informação atualizada para todos os servidores. Isto requer troca de mensagens entre os servidores que estão ligados remotamente numa árvore. O algoritmo que foi utilizado para replicação de informação foi o seguinte:

Ao receber uma atualização, o servidor envia ao seu pai uma primeira mensagem com a região e o tamanho dos dados, e numa segunda mensagem os dados. Este pedido é recebido pelo pai que está à espera das duas mensagens e após as receber envia o pedido para o seu pai. O pedido de atualização subirá na árvore enquanto não chegar à raiz da árvore. Ao chegar à raiz, o pedido é enviado para um *pipe*,

que sequencializa os pedidos. A raiz lê do textitpipe os pedidos, faz a atualização dos dados na região e envia para todos os filhos os dados da atualização (primeira mensagem com informação da região e do tamanho dos dados, segunda mensagem com os dados). Os filhos estão à espera de receber as duas mensagens numa thread e quando as receberem enviam-nas para cada um dos filhos. O processo de descida de informação irá ocorrer até a informação chegar aos servidores que não têm filhos, que apenas atualizam os dados na região.

4 Threads utilizadas no programa

Nas seguintes secções apresenta-se um fluxograma para cada uma das *threads* criadas, com uma breve explicação do que cada uma faz.

4.1 Wait_clipboard

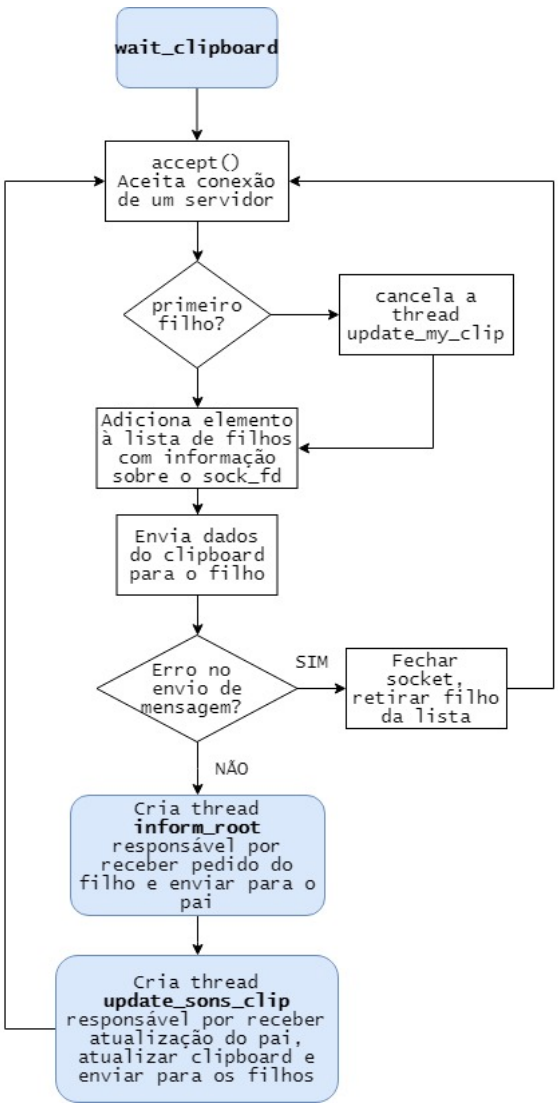

Figura 5: Fluxograma da *thread* que contém função wait_clipboard

Na figura 5 encontra-se o fluxograma da *thread* que espera por conexões de outros servidores. Na primeira conexão recebida, o servidor cancela a *thread* anteriormente criada, *update_my_clip*. Ao aceitar um servidor, envia-se os dados do clipboard para o filho e se não ocorrer nenhum erro durante o envio dos dados ou na alocação de memória dos *buffers* criam-se duas *threads*: a *inform_root* que é responsável por ler pedidos do filho e enviá-los para o pai, e a *update_sons_clip* que é responsável por ler os dados a serem atualizados do pai, atualizar o *clipboard* e enviar para todos os filhos presentes na lista de filhos.

4.2 Inform_root

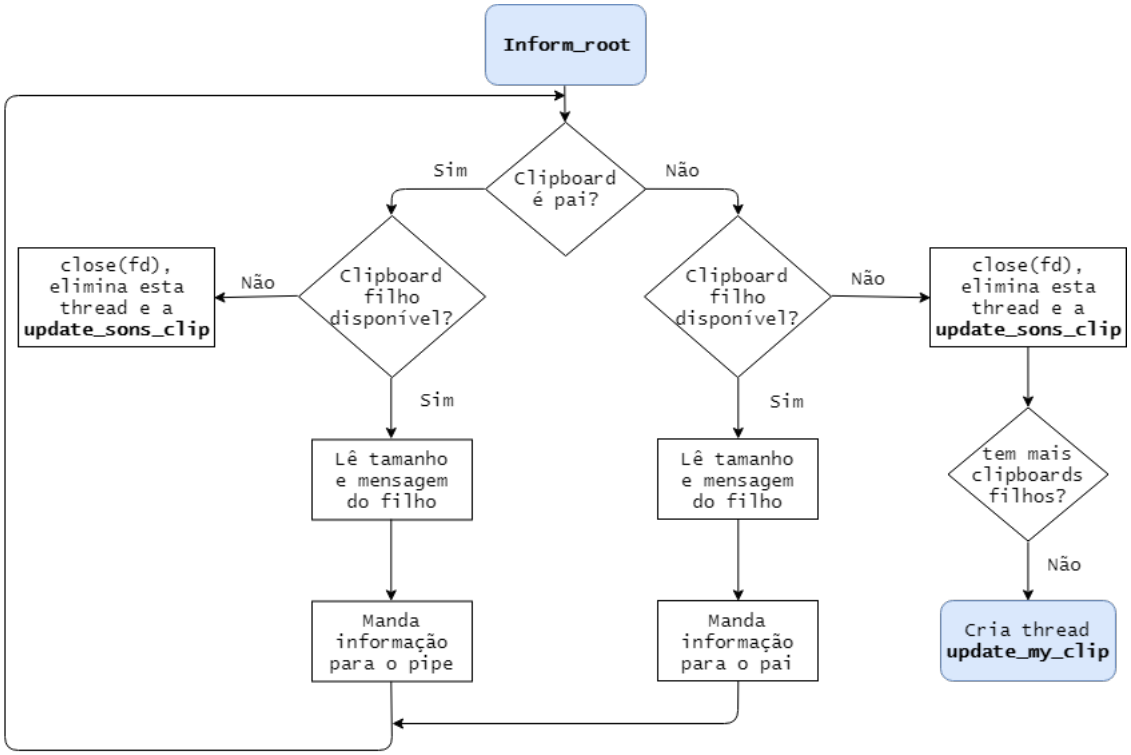

Figura 6: Fluxograma da *thread* que contém função inform_root

Na figura 6 encontra-se o fluxograma da *thread* que é responsável por ler pedidos do filho e enviá-los para o pai. Existem 2 caminhos possíveis: se o servidor não tiver pai (é *root*), isto é, foi criado em modo singular, se o filho fôr encerrado fecha-se o socket dedicado a essa comunicação e eliminam-se as *threads update_sons_clip* e *inform_root*. Se não for encerrado lê o tamanho e os dados (pedido de atualização) do filho e envia a informação para o *pipe*. Caso o servidor tiver pai, se o filho fôr encerrado fecham-se as *threads* tal como no caso anterior, e caso o servidor ficar sem filhos cria-se a *thread update_my_clip* que vai ser explicada de seguida. Se o filho não encerrar é lido o tamanho e os dados (pedido de atualização) do filho e enviada a informação para o pai.

4.3 Update_sons_clip

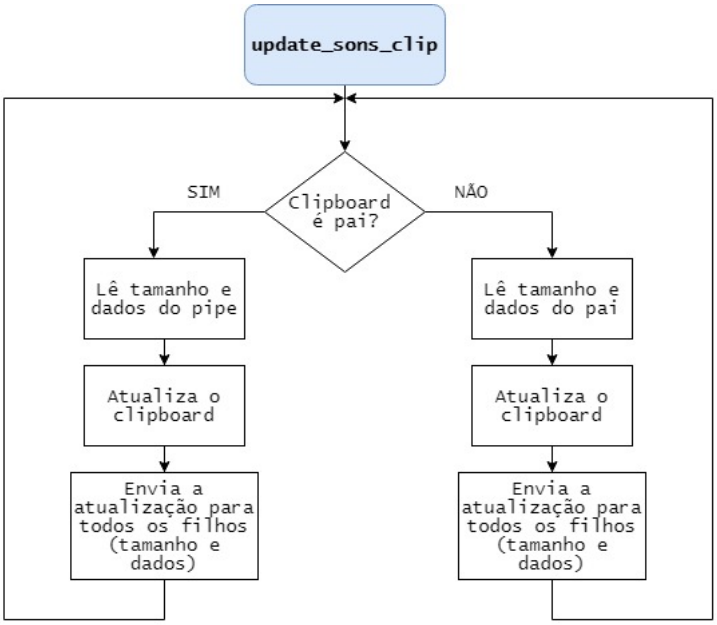

Figura 7: Fluxograma da *thread* que contém função update_sons_clip

Na figura 7 encontra-se o fluxograma da *thread* que é responsável por ler os dados a serem atualizados, do pai, atualizar o *clipboard* e enviar para todos os filhos presentes na lista de filhos. Existem 2 caminhos possíveis: Se o servidor fôr a *root* lê os dados do *pipe*, atualiza o *clipboard* e envia a informação para todos os filhos. Caso não seja a *root*, a única diferença é que em vez de ler de um *pipe* lê do pai e os passos seguintes são os mesmos.

4.4 Update_my_clip

Na figura 8 encontra-se o fluxograma da *thread update_my_clip*. Esta *thread* é criada quando o servidor se conecta a outro e é responsável por ler informação desse outro e atualizar o seu *Clipboard* quando ocorrer uma atualização de uma região em qualquer servidor da árvore. Caso não seja possível estabelecer a comunicação com o *Clipboard* a que se ligou, elimina-se a ponte de conexão e este servidor passa a ser *root* de uma nova árvore.
Quando um novo *Clipboard* conecta-se a este esta função vai ser eliminada. Pois a comunicação entre servidores adjacentes passa a ser feita pelas *threads inform_root* e *update_sons_clip* criadas pela *thread wait_clipboard*

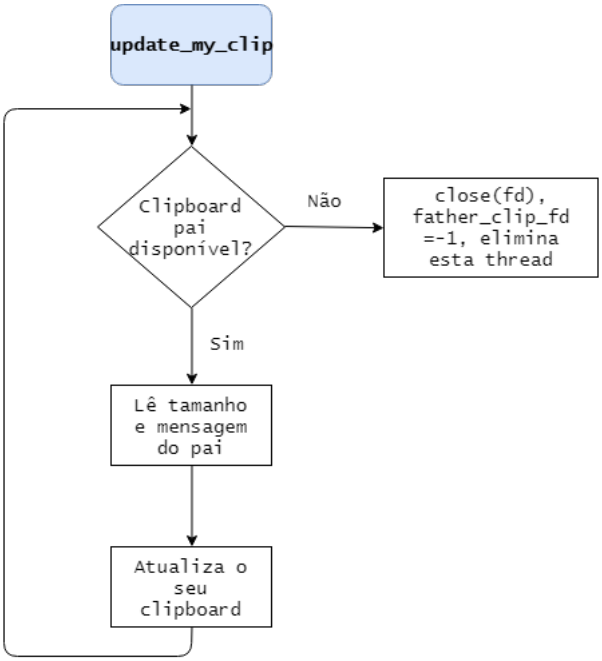

Figura 8: Fluxograma da *thread* que contém função update_my_clip

4.5 Data_processing

Na figura 9 representa-se o fluxograma da *thread data_processing*. Esta *thread*, é responsável pela comunicação entre um cliente e o servidor. Portanto, é ela que gere os pedidos de cada aplicação e envia a resposta para o cliente com a informação de acordo com o pedido. Sempre que, existe uma nova aplicação a conectar-se ao servidor esta *thread* vai ser criada. Consequente, existem tantas *threads* deste tipo como aplicações ligadas ao servidor.
Esta função encontra-se num loop infinito à espera de mensagens por parte da aplicação. Se a ligação entre a aplicação e o servidor for quebrada, o servidor deixa de receber pedidos do cliente e esta *thread* é eliminada. Quando recebe um novo pedido está função vai tratar dele de acordo com a ação a realizar, se por algum motivo pedido foi inválido não será executada nenhuma ação nesta iteração do ciclo.

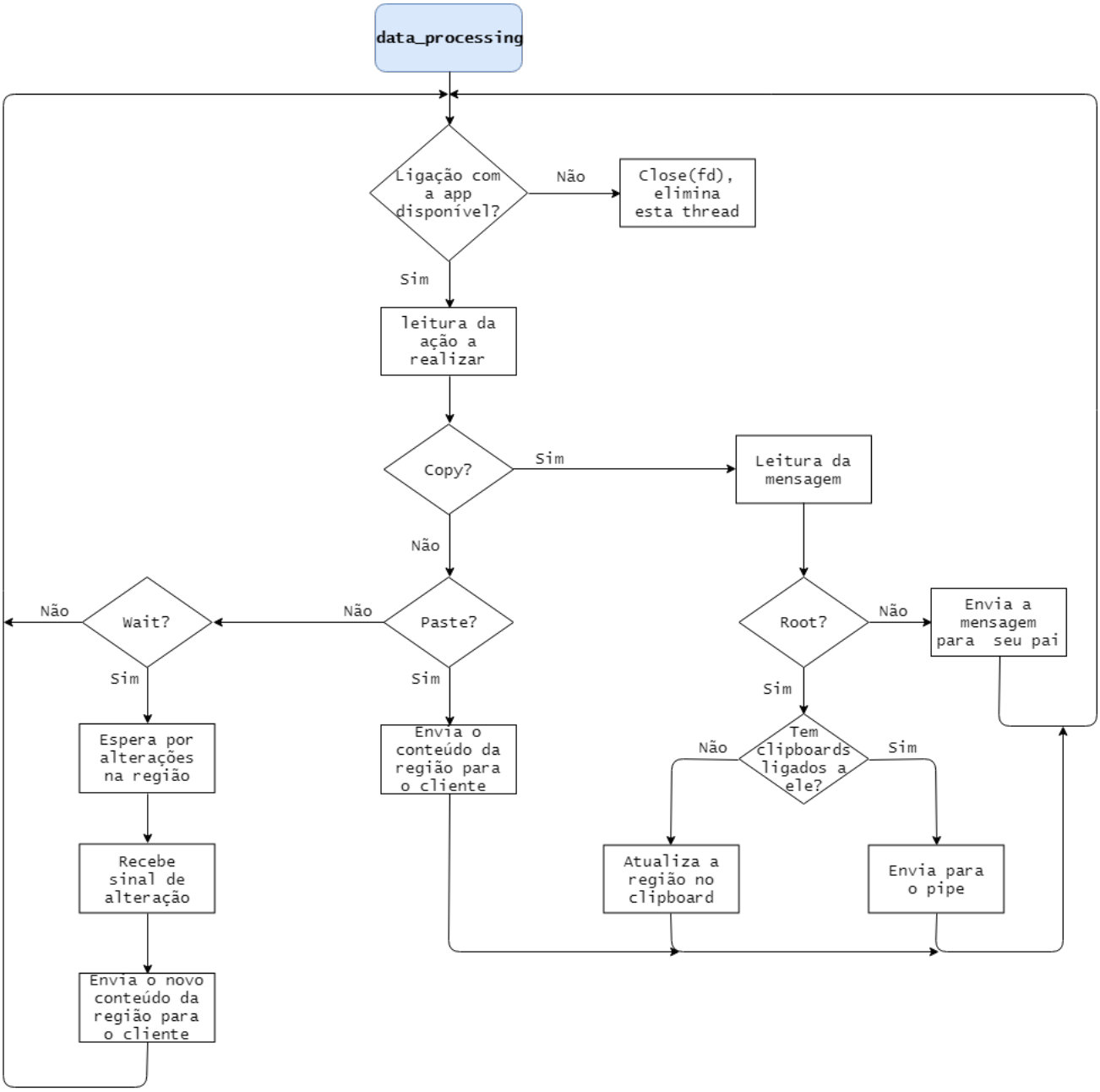

Figura 9: Fluxograma da *thread* que contém função data_processing

5 Sincronização

Uma vez que este é um sistema *multithreaded*, em que existem estruturas de dados partilhadas entre várias *threads*, é necessário prevenir acessos e escritas por diferentes *threads* em simultâneo, uma vez que nesse caso poderá obter-se informação inválida. Para resolver este problema de sincronização, começou-se por identificar as regiões críticas e de seguida implementou-se a exclusão mútua recorrendo ao mecanismo que se achou mais adequado.

5.1 Identificação de regiões críticas

Uma região crítica é a região do código que engloba um conjunto de instruções que quando realizadas em paralelo com outras *threads*. podem provocar erros ou comportamentos inesperados. No nosso projeto detetaram-se as seguintes regiões críticas:

1. Leituras e escritas na estrutura de dados responsável por armazenar os dados das 10 regiões.
2. Envio de pedidos/mensagens de vários filhos para o mesmo servidor em simultâneo (tem de se garantir que a primeira mensagem com o tamanho e região e a segunda com os dados são lidas consecutivamente pelo pai).
3. Acessos e escritas nos ponteiros relacionados com a lista de filhos de cada servidor.

5.2 Implementação da exclusão mútua

Para se resolver o primeiro problema de sincronização percebeu-se que não existe problema caso estejam várias threads a ler/aceder à estrutura de dados responsável por armazenar os dados das 10 regiões. Apenas existe problemas caso ocorram leituras e escritas em simultâneo, ou escritas em simultâneo. Por este motivo, recorreu-se à implementação de *read/write locks*. Sempre que alguma das regiões é alterada bloqueia-se a região para escrita na porção de código responsável. Sempre que se lê ou se acede a uma região do *clipboard*, bloqueia-se a porção de código correspondente para leitura.

Na segunda região crítica identificada percebeu-se que quando um pai lê dois pedidos de atualização de dados de dois filhos em simultâneo, pode haver problema se as duas mensagens enviadas por cada filho (região e tamanho; dados) não forem lidas pelo pai consecutivamente, isto é, se o pai ler por exemplo o tamanho e região do filho 1 e de seguida ler a região e tamanho do filho 2, em vez de ler os dados do filho 1. Este problema resolveu-se recorrendo a um *mutex* e bloqueou-se a porção de código responsável por fazer escritas no pai.

O terceiro problema de sincronização ocorre, por exemplo, quando dois servidores se conectam no mesmo instante ao mesmo servidor, uma vez que no processo de um servidor se ligar a outro, há a alteração da lista de filhos. Deste modo o processo de adição de filhos à lista tem de estar bloqueado através de um *mutex*, de modo a apenas se adicionar o segundo após a adição do primeiro estar concluída. Além disto, existe alteração dos ponteiros da lista no envio de informação para todos os filhos e quando um servidor se desconecta da rede, pelo que tem de ser removido da lista de filhos do seu pai. Bloquearam-se todas as porções de código responsáveis por estas funcionalidades.

6 Tratamento de erros

Para correto funcionamento do programa é necessário prever e escrever código que lide com possíveis utilizações anómalas por parte dos clientes, tornando o sistema

robusto e mais imune a possíveis "ataques".

6.1 Tratamento de erros do lado do servidor

O servidor pode ser iniciado de duas formas diferentes: se for conectado em modo singular, escreve-se na linha de comandos "./clipboard", caso seja em modo conectado, escreve-se "./clipboard -c <endereço_ip> <porto>". Para tratar más inicializações do servidor, mostra-se uma mensagem de erro ao utilizador a explicar o modo de inicialização sempre que *argc* (número de argumentos) for diferente de 1 e de 4, seguida do encerramento do programa através de um exit com valor -1 (sempre que se encerra o programa devido a tratamento de erros retorna-se o valor -1). Caso *argc* for diferente de 4, se o 2º argumento fosse diferente de -c mostra-se a mesma mensagem de erro e fecha-se o programa.
Se o servidor for iniciado em modo conectado e cria-se um socket para comunicar com o pai e estabelece-se a conexão. Se nalgum destes passos ocorrer um erro, Imprime-se uma mensagem de erro e fecha-se o servidor, visto que este não conseguiu ligar-se à rede, como pretendido. De seguida o servidor recebe do pai os dados das 10 regiões do *clipboard*. Durante este processo, se houver problema na receção dos dados imprime-se uma mensagem de erro e fecha-se o programa.
Durante a criação do *socket* para receber conexões por parte de outros *clipboards*, se houver um problema na criação do socket encerra-se o servidor. Se houver problema a atribuir um endereço para um ponto de comunicação com um servidor (*bind*), volta-se a tentar e quando se conseguir retorna-se o descritor de ficheiro do *socket*. A partir deste momento cria-se uma *thread* responsável por receber conexões de outros servidores. Se ocorrer algum problema ao aceitar a conexão de um servidor imprime-se uma mensagem a informar e salta-se para a próxima iteração onde se espera a conexão de mais servidores (o mesmo acontece na *thread* que espera conexão de aplicações). Quando se estabelece a conexão, ao adicionar o filho na lista, se houver erros a alocar memória o programa é encerrado. Ao enviar-lhe os dados do *clipboard*, se houver erro no envio de mensagens o programa é também encerrado. Nas *threads* que foram criadas para estabelecer comunicação com os servidores adjacentes, caso o valor retornado pelas funções *write()* e *read()* seja 0, presume-se que o servidor adjacente foi encerrado e faz-se um conjunto de instruções que processa esse acontecimento. Na *thread* responsável por tratar dos pedidos do cliente, sempre que o envio ou a receção de uma mensagem seja anómala, imprime-se uma mensagem de erro e encerra-se o servidor. É de referir que surgiu a necessidade de tratar o sinal SIGPIPE, ignorando-o, de modo a qu o programa principal não se fechasse inesperadamente.

6.2 Tratamento de erros do lado do cliente

Ao iniciar uma aplicação, cria-se um *socket* para comunicar com o servidor e estabelece a conexão com o servidor através do *socket*. Se surgir algum problema durante a criação do *socket*, ou durante o estabelecimento da conexão, a aplicação é fechada. Nas funções responsáveis pela implementação das ações "copiar", "colar"e "esperar", verifica-se primeiramente se a região pedida pelo utilizador está compreendida

entre 0 e 9 e caso não esteja imprime-se uma mensagem de erro na consola e retorna-se o valor 0 em vez de um número positivo de *bytes*. Se o ocorrer problemas durante o envio de mensagens para o servidor ou durante a receção, retorna-se também o valor 0. Se o valor retornado por uma destas funções for 0 a aplicação criada imprime uma mensagem de erro ao utilizador a informá-lo do erro.

7 Gestão de recursos

Na realização de um projeto, um dos principais objetivos é controlar ao máximo o consumo de recursos da máquina. Pretende-se assim, que a memória gasta pelo sistema seja a mínima possível e que a utilização do processador não esteja acima do normal.
Com o objetivo de consumir memória o mínimo possível, optou-se em utilizar uma estrutura para armazenar a informação de tamanhão variável, ou seja, para cada espaço do *Clipboard* aloca-se um espaço de memória do tamanho da mensagem que vai ser guardada. Caso se pretenda guardar informação num local preenchido, a informação inicial será eliminada pela função *free* e o tamanho da região será ajustado à nova mensagem. Como já foi dito anteriormente, existe uma lista para guardar a informação dos filhos conectados, quando conecta-se um novo *Clipboard* aloca-se memória para um novo nó da lista. Em sentido inverso, quando um *Clipboard* se desconecta o correspondente nó na lista vais ser eliminado através da função *free*. Caso tenha *Clipboards* ligados a ele, as *Threads inform_root* e *update_sons_clip* para comunicar com cada um deles serão eliminadas.
Do lado da aplicação, quando um cliente recebe uma mensagem do servidor está vai ser guardada num *buffer* auxiliar com o tamanho da mensagem recebida. Uma vez que, o *buf* pode não ter tamanho suficiente para guardar a mensagem completa, esta vai ser cortada. No final da aplicação, a memória de ambos é libertada.
Relativamente aos *Sockets*, a sua gestão é feita através das funções *unlink()* e *close()*. Quando um *Clipboard* desconecta-se, o *socket* responsável pela comunicação com o cliente é eliminado pelo *unlink()*. Esta função remove o *socket* do sistema de ficheiros. No entanto, quando estamos em rede e por algum motivo um servidor fica inativo, o *socket* responsável pela ponte e eliminado pela função *close()*.