C:\Users\Pedro Tomás\Documents\IST\logo_IST_A.emf

**Clipboard distribuído**

Programação de Sistemas

**Alunos:**

Nome: André Cavalheiro Número: 84000

Nome: John Mendonça Número: 84106

Nome do Docente: João Nuno De Oliveira e Silva

**Índice**

[1 Arquitetura 3](#_Toc516171508)

[1.1 Comunicação Local 3](#_Toc516171509)

[1.2 Comunicação Remota 4](#_Toc516171510)

[2 Gestão de Threads 4](#_Toc516171511)

[3 Sincronização 4](#_Toc516171512)

[3.1 Clipboard 4](#_Toc516171513)

[3.2 Envio de dados para o exterior 5](#_Toc516171514)

[*3.3* Criação de *threads* 5](#_Toc516171515)

[4 Conclusões Finais 5](#_Toc516171516)

# Arquitetura

O programa pode ser divido em dois blocos principais, cada um deles com sub-blocos com propósitos específicos. Estes blocos são inicializados no início da execução e são independentes entre si.

## Comunicação Local

A comunicação remota entre os utilizadores e o próprio clipboard é feita a partir de sockets da família AF\_UNIX e do tipo SOCK\_STREAM. A escolha entre os diversos tipos de sockets foi feita levando em consideração a natureza dinâmica das mensagens a ser transmitidas.

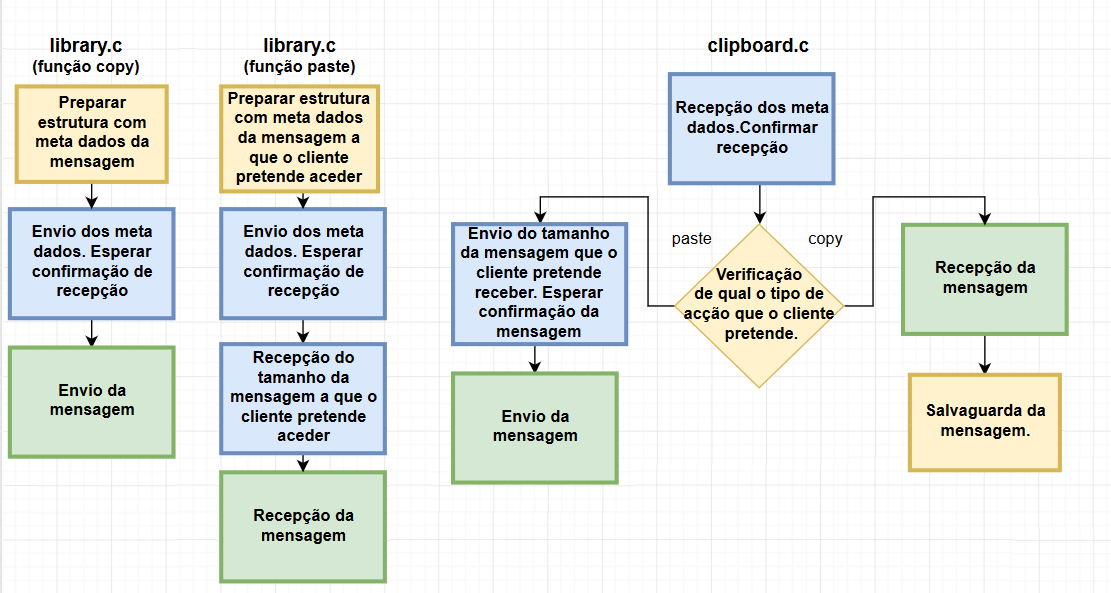
Tendo isto em mente o protocolo de comunicação entre os dois programas segue o seguinte padrão:

* É feita uma primeira comunicação para informar ambos os lados da informação necessária à transmissão da mensagem. Este diálogo ocorre sempre com o uso da estrutura representada na figura …, a partir das funções *handShake* e *handleHandShake*.
* A mensagem é transmitida de um lado para o outro a partir de um ciclo while, garantindo sempre o envio total da mensagem ou o lançamento de um erro. Esta transmissão é feita usando ponteiros do tipo *void\** e as funções *sendData* e *receiveData*.

Meter aqui print da estrutura struct metaData.

Os diagramas representativos das funções mais importantes na comunicação local encontram-se representados na figura ….

* Os retângulos representados a azul representam as funções handShake / handleHandShake consoante se trata do envio dos meta dados ou receção dos mesmos respetivamente.
* Os retângulos a verde representam as funções *sendData* */* *receiveData* consoante se trata do envio da mensagem ou da receção da mesma respetivamente.
* Os retângulos a amarelos não representam qualquer função, mas sim passos intermédios e necessários ao funcionamento do programa.



(falar aqui de como é lançada uma thread para cada usuário local do clipboard ou isso fica tudo explicito na secção 2?)

## Comunicação Remota

A comunicação remota entre clipboards e a primeira funcionalidade a ser lançada no sistema, dado que este é um parâmetro de entrada e poderá ser necessário obter informação localizado num clipboard remoto. Portanto, é necessário estabelecer todas as ligações necessárias entre clipboards antes de se proceder ao lançamento do bloco que lida com as comunicações locais.

Como referido anteriormente, é possível ser dado um parâmetro de entrada que corresponde à localização internet do clipboard a quem nos queremos ligar. Este clipboard estará a um nível superior ao nosso, se considerarmos o conjunto das ligações como uma árvore.

É realizada uma coneção com o clipboard e é realizada uma sincronização da informação. A extração do conteúdo do clipboard é realizada utilizando a função *ClipSync()*. Dada a relevância desta função esta é executada em série com o programa(não é lançada nenhuma *thread* antes, e a função é executada pela *thread* principal). Após estabelecida a primeira sincronização é lançado uma *thread* cujo responsabilidade é lidar com a comunicação entre o clipboard e o clipboard exterior. Esta executa a função *handleClipboard()* e mantêm-se em execução até fecho de ligação(quer local, quer exterior).

No caso de não haver indicação de ligação exterior, ou após o lançamento da função que lida com o “pai”, é lançado um *thread* que executa a função “ClipHub”. Como o nome dá a entender, esta função pode ser considerada como o ponto de convergência de todas as ligações, visto que é esta que trata todas as ligações subsequentes entre clipboards. É nesta função que se abre as ligações ao exterior e lança novas thread que executam a função *handleClipboard()* a cada nova ligação, bem como a adicionar à lista de ligações.

A função handleClipboard() é responsável por diversas tarefas dentro do âmbito das ligações entre clipboards, visto que é ela que trata da ligação com um clipboard em específico. Em primeiro lugar, dado que cada ligação é única e exclusiva, na medida em que um clipboard só se tenta ligar a outro uma única vez, a função handleClipboard() comunica com o clipboard exterior de modo a enviar toda a informação relativa ao clipboard atual.

Após a conclusão da sincronização a função aguarda por nova informação proveniente do exterior. Se a informação recebida for nova( ou seja, se a hash for diferente da já presente no clipboard) esta é atualizada no clipboard e adicionada à lista de replicação.

# Gestão de Threads

# Sincronização

O Clipboard Distribuído tem no decorrer da sua execução diversas *threads* que necessitam de aceder e/ou alterar a mesma informação. De modo a garantir o funcionamento pretendido do programa será necessário garantir a sincronização das mesmas, de modo a evitar problemas (como por exemplo as *race conditions)* e manter a integridade dos dados.

## Clipboard local

Podemos considerar o Clipboard Distribuído na sua essência como um problema de Produtor-Consumidor, em que diversas entidades (*threads*) necessitam de aceder a uma dada região do Clipboard para ler(*paste*) ou escrever (*copy*). Visto que se pretende manter coerência no conteúdo, será necessário garantir que o processo de *paste* não interfira com o processo de *copy* e vice-versa e, portanto, estamos perante uma zona crítica.

O clipboard está subdivido em várias regiões, cada uma com a sua própria região crítica. Esta região crítica está, por sua, vez implementada utilizando dois métodos em simultâneo: o *read-write* *lock* e o *mutex*. Tratando-se de um problema de produtor-consumidor, o tipo de implementação mais adequado para esta região crítica é o *read-write lock*, pois permite a leitura do conteúdo por parte de várias *threads* em simultâneo. No entanto, de modo a acomodar a função *wait*, que é um requisito do cliente local, bem como das conexões exteriores (de modo a manter a sincronização global) foi necessário introduzir um mutex, de modo a podermos utilizar a implementação POSIX da espera condicional. A utilização da espera condicional de novos dados em detrimento da espera ativa (*while() loop*) conduz a uma maior eficiência do projeto. Na figura seguinte está ilustrado a zona crítica de uma região do clipboard.

## Envio de dados para o exterior

Após aquisição de novos dados, que são colocados numa lista de espera para envio (que constitui, por si só, uma região critica protegida por um *mutex*), a *thread* dedicada ao envio de nova informação ao exterior da dada região necessita de percorrer uma lista ligada dedicada para o efeito. Esta lista, sendo partilhada entre as threads das várias regiões, constitui uma região crítica, pois podem ocorrer novas ligações ou fecho de outras em tempo real. Deste modo, será necessário proteger esta região com exclusividade(*mutex)* pois todos os processos com acesso a esta lista têm a possibilidade de eliminarem elementos da lista, caso contrário um *read-write lock* seria o ideal. O mecanismo suplementar de atualização da lista após o fecho de uma ligação consiste na *thread* responsável pelo envio da informação eliminar o elemento quando deteta no envio pelo *socket* um erro.

## Criação de *threads*

Ao longo de todo o processo existem situações em que são necessários criar *threads* semelhantes, nomeadamente as *threads* de envio de informação ao exterior e de *handling* de conexões com clientes locais, em que estas reutilizam as mesmas variáveis (pois elas só contêm informação necessária para a *thread*) e, portanto, existe uma *race condition*. De modo a evitar este problema, utilizou-se um *mutex* comum para todo o processo. Cada nova thread só liberta a zona crítica quando tiver todos os parâmetros em questão em memória própria.

## Sincronização global

Como indicado anteriormente, dado que o Clipboard distribuído consiste num problema de produtor-consumidor, é necessário garantir integridade da informação presente no Clipboard distruibuído. Quer isto dizer que a informação presente em cada clipboard da rede em repouso(ou seja, na ausência de qualquer introdução de nova informação no Clipboard) tem de ser coerente, igual. De igual forma, será benéfico que a informação presente no Clipboard seja a mais recente possível, pois é o que faz sentido num contexto de utilização em tempo real. Desta forma foram utilizados diversos mecanismos, quer locais, que globais de modo a cumprir as premissas acima mencionadas. De modo a facilitar a explicação da sincronização global, optou-se por uma nomeculatora de teoria de árvores.

D modo a garantir que em repouso a informação seja coincidente, será necessário garantir que qualquer introdução de novos dados num dado clipboard seja replicado para os restantes elementos da rede. Desta forma, qualquer novo copy realizado num dado clipboard, que inclui introdução global, ou até mesmo provenientes do exterior, é replicado para o pai e para os filhos, salvo a exceção de a informação ter vindo do pai. Tal exclusão é necessária visto que assume-se que informação proveniente de cima é considerada mais recente por parte o pai, e portanto a não inclusão deste caso implicaria em instabilidade do sistema dado que nesta situação entraríamos num *loop* infinito de replicação pai-filho. Este método de manutenção do sincronismo pode ser considerado de força bruta, dado que existe grandes fluxos de replicação, que por sua vez podem ser anulados por um *copy* subsequente. No entanto, a utilização de uma impressão digital única para cada informação, denominado *hash* facilita neste processo, na medida em que informação recebida dos filhos, cuja origem é de um nível superior ao dado filho(o que implica que informação passou pelo nó atual, e replicado para baixo, os tais filhos) não é enviada. Este processo é implementado através de um *handshake* (teoria de comunicação), que informa ao recetor a *hash* dos dados que se pretendem enviar. Caso a hash coincida, conclui-se que a informação já está no Clipboard e portanto o envio *bytestream* dos dados é descartado.

Apesar de não ser garantido que a informação que se encontra presente é a mais recente após todas as iterações de replicação se encontrarem concluídas, esta implementação permite que os clipboards adjacentes ao foco da nova informação tenham acesso à informação mais recente dos seus vizinhos. Contrastando com esta implementação, foi considerado uma outra implementação que consistia no envio de qualquer nova informação ao topo da árvore, e esta seria responsável por atualizar a restante árvore, considerando a última informação a chegar ao topo a mais recente. Ora apesar de esta implementação também garantir estabilidade e coerência em todos os clipboards, um *paste* imediatamente após um *copy* numa dada zona não iria retornar o *copy* acima mencionado, o que do ponto de vista do problema de Clipboard Distribuído não faz muito sentido.

# Conclusões Finais

De maneira geral foram cumpridos todos os requisitos pretendidos para o programa Clipboard Distribuído. Em baixo encontram-se algumas explicações acessórias e justificações de não implementação, se se tratar do caso.

## Manutenção

Um erro de desenvolvimento do projeto foi não ter em consideração na arquitetura do projeto a manutenção do sistema em termos de libertação de memória. Dada a complexidade do programa, que inclui várias threads com memória própria, desde buffers até estruturas, foi dificil encontrar um mecanismo adequeado que garantisse a total libertação de memória. A principal dificuldade encontrada foi a de informar threads que se encontravam bloqueadas, quer na espera condicional, quer na leitura de comunicação exterior.

Foi considerado a utilização de uma message queue, ou PIPE, que representasse um sinal de encerramento para as threads. Na parte das threads seria implementado um poll() que escutasse ao FD do sinal bem como aos outros necessários. De modo a garantir este funcionamento seria necessário utilizar os leitores em modo *non-block*. O *accept4()* é a única implementação que aceita este tipo de flag, no entanto dado que esta função não é nativa de Linux, esta opção foi descartada.

## Error handling

Ao longo de todo o código existem diversas instâncias de error handling, sendo estes divididos em fatais e não fatais. Podemos considerar um erro fatal aquele que afeta o funcionamento essencial do programa, na medida em que a presença deste tipo de erro é indicação de um problema grave e que a continuação do programa não é garantida. Exemplos deste tipo incluem erros em alocação de memória, libertação de nodos da lista, entre outros. Os erros não fatais são aqueles que, apesar de indicarem um problema, não são impeditivos da continuação da execução do programa. A maior parte dos erros de leitura e escrita(conecção) para o exterior são englobados neste tipo, dado que na ocorrência deste erro, o programa simplesmente desliga a conecção e resume a execução.

## Fechos de ligação

Na ocorrência de um fecho de um clipboard, é informado a todos os clipboards ligados que irá ocorrer um shutdown. Neste caso, os recetores procedem a diversos procedimentos para garantir a sua execução livre de erros, nomeadamente a remoção em nós de listas e fecho de sockets. Acontece de forma semelhante entre o cliboard e o cliente.

Foi testado com sucesso diversos casos, nomeadamente o fecho de clientes, de filhos e pais. É de notar que no caso do pai se fechar, o clipboard continua a correr em modo autónomo, na medida em que ele passa a ser o pai da nova árvore.

