INSTITUTO SUPERIOR TÉCNICO

Programação de Sistemas

Relatório do Projecto

Clipboard Distribuído

81282 – Luís Simões

81216 – Bernardo Amaral

Prof. João Nuno De Oliveira e Silva

Lisboa, 05 de junho de 2018

Índice

1. Introdução

O objetivo deste projeto é a realização de um sistema de comunicação entre aplicações através de servidores distribuídos e sincronizados remotamente, onde as aplicações podem guardar mensagens no correspondente servidor local, que serão replicadas em todos os servidores remotos na mesma rede, ou ler informações que estejam inseridas no servidor previamente, ou ainda verificar se alguma posição de memória no servidor é alterada.

Os servidores (*clipboards*) devem estar interligados em rede (neste caso numa topologia em árvore) de maneira a que quando um servidor recebe uma mensagem, vinda de uma aplicação, esta é replicada para todos os outros servidores, de modo a que todos os servidores tenham sempre exatamente a mesma informação, ou seja, estejam em sincronismo.

2. Arquitetura do Sistema

A arquitetura pretendida para este sistema é a descrita na seguinte figura:

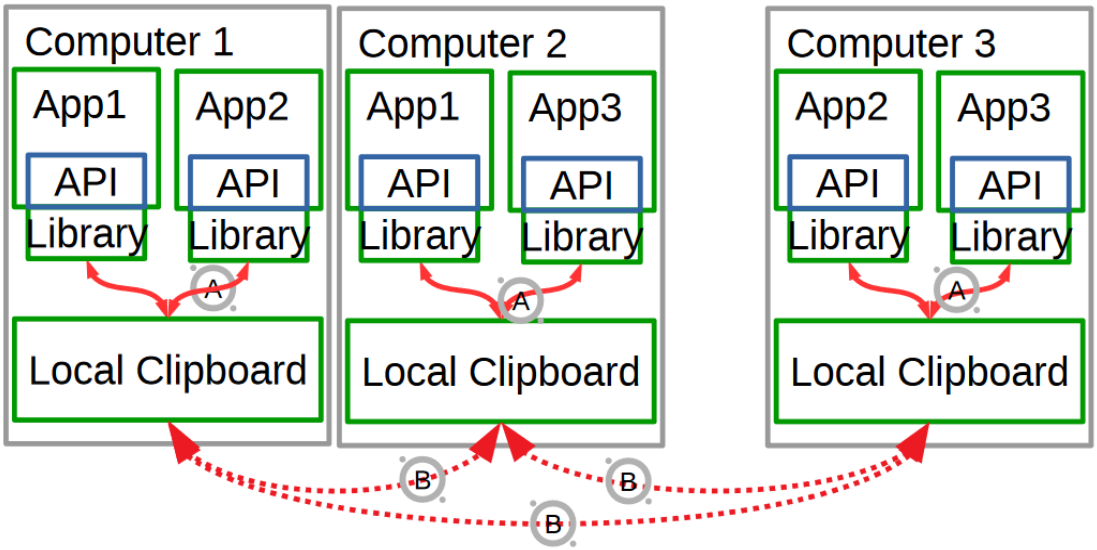

Figura 1 - Arquitetura do Sistema de Comunicação

O sistema apresentado é composto pelos seguintes módulos:

- **API –** A interface do utilizador, que contém apenas a informação estritamente necessária para que os utilizadores, cujo objetivo é desenvolver aplicações baseadas na comunicação entre **clipboards**, consigam integrá-los facilmente nas suas aplicações. No projeto realizado, a **API** é constituída pelo ficheiro *clipboard.h* que apenas contém o cabeçalho das funções da **Library**.

Deste modo garante-se que o cliente esteja abstraído do que se passa para além da aplicação que está a desenvolver.

- **Library –** No projeto realizado, é constituída pelo ficheiro *library.c*, que consiste no código desenvolvido para as funções cujo cabeçalho está na **API**. Este ficheiro é o responsável por tratar dos pedidos das aplicações, fazendo as comunicações necessárias com o servidor.

- **Local Clipboard –** É o servidor local do sistema, constituído pelo ficheiro *clipboard.c*, que comunica tanto com as aplicações locais, como com os servidores que se encontram distribuídos por outros computadores e entre os quais há uma ligação. Cada **Local Clipboard** é responsável por guardar e replicar a informação de maneira sincronizada entre os clipboards que constem da mesma rede. Cada clipboard é ainda composto por dez regiões, nas quais podem ser guardados qualquer tipo de dados.

2.1. Protocolo de Comunicação

A partir do esquema apresentado na figura 1, verifica-se que no sistema existem dois protocolos de comunicação diferentes. A comunicação entre a aplicação e o servidor local (A), e a comunicação entre dois servidores da mesma rede (B).

Estes protocolos são diferentes na medida em que o primeiro representa uma comunicação local (A), enquanto que o segundo representa uma comunicação remota, através da internet (B). No entanto, o protocolo utilizado é semelhante em ambos, pois no sistema implementado são usadas **Sockets** como meio de comunicação, em ambas as situações.

- **Comunicação local –** De todos os tipos de canais de comunicação lecionados na disciplina de Programação e Sistemas, foi na comunicação local que havia mais margem de manobra para escolher o protocolo de comunicação, no entanto a escolha recaiu em **Sockets** (domínio *UNIX*) em vez de **FIFO's**, ou de **Pipes**, pois a comunicação entre a aplicação e o servidor é bidirecional, coisa que os **Sockets** permitem fazer de forma direta, enquanto que os restantes são canais de transmissão unidirecionais.

- **Comunicação entre Servidores –** Neste caso, foi necessário a utilização de **Sockets**. A principal diferença em relação à comunicação local reside no facto de que, entre servidores, os **Sockets** são de domínio *INET* em vez de *UNIX*, pois estes permitem uma comunicação através da internet, enquanto que os outros apenas permitem comunicação a partir do *kernel* local.

Visto que através de sockets apenas é possível enviar e receber mensagens por estruturas do tipo *void* *, de modo a que a comunicação entre serviços seja eficiente, foi criada uma estrutura de dados própria para envio, a *struct _data*. Esta contém todas as informações necessárias à especificação e

tratamento de pedidos, e é transformada em *bytestream* antes de ser enviada, e convertida de volta para estrutura *_data*, no momento da receção. Esta estrutura contém a região sobre qual é feito o pedido, o tipo de pedido que se pretende executar, o tamanho do buffer, caso seja necessário enviar alguma informação, e um vetor que contém a informação que se quer enviar.

2.2. Funcionamento das *Threads*

Para que o servidor local possa estar simultaneamente em comunicação com várias aplicações, servidores, e ainda a aceitar novas ligações de ambos os tipos, é necessário que o tratamento de todas estas tarefas seja dividido. Para tal foram utilizadas **Threads.**

A decisão tomada quanto à divisão de tarefas teve apenas em atenção quantas operações é que devem estar em execução ao mesmo tempo. Para tal, de acordo com a enumeração no parágrafo acima, foi necessário dividir inicialmente a execução do programa em dois processos, através da criação de uma **Thread**, em que um processo aceita novas ligações de aplicações, enquanto que o outro aceita novas ligações de outros servidores. O processo que aceita aplicações encontra-se na função *main*, poupando assim algum processamento desnecessário, enquanto que o que aceita servidores executa a função *remote_connect*.

Estes dois processos executam essencialmente o mesmo tipo de operações. Ambos se encontram num ciclo infinito onde estão constantemente a aceitar novas ligações, e por cada ligação estabelecida, é criada uma nova **Thread**. Estas **Threads**, serão responsáveis pela comunicação com cada um dos serviços ligados, ou seja, com cada uma das aplicações e cada um dos servidores com os quais foi estabelecida uma ligação, e para que esta comunicação seja "contínua" e mais simples, decidiu-se que cada nova ligação estabelecida com o servidor local, seria tratada apenas por uma **Thread**, só para tratar dos seus pedidos e comunicações. Assim, criou-se uma função *app_receiver* que trata das comunicações entre uma aplicação e o servidor local, recebe os pedidos da aplicação e executa esses pedidos adequadamente. E criou-se também outra função, *clipboard_receiver*, que trata de atualizar ambos os servidores, o remoto e o local, e é responsável pelo sincronismo entre eles, de modo a terem sempre a mesma informação.

2.3. Estruturas de Dados Partilhadas

Visto que são utilizadas inúmeras **threads** no programa que executa o servidor, e que tem de haver informação partilhada entre estas, é necessário a

definição de variáveis e estruturas de dados que sejam acessíveis sincronizada e simultaneamente pelos processos a decorrer. Tal é necessário, visto que todos os processos terão necessidade de aceder às mesmas variáveis atualizadas, mesmo que estas tenham sido modificadas após a sua execução, por outras threads.

Para tal foram usadas variáveis globais, estritamente para os casos em que estas eram necessárias. As variáveis globais usadas foram as seguintes:

- *struct _region regions[10]* : Vetor que guarda todas as regiões e respetivo conteúdo, no servidor local. É necessário defini-lo como global pois há várias threads diferentes que necessitam de lhe aceder, e ele tem de estar igual em todas estas. Cada região é constituída por uma variável do tipo *size_t*, que contém o tamanho da informação armazenada nessa região, e um buffer do tipo *void* * que contém a informação armazenada, o seu tipo possibilita que seja guardado qualquer tipo de informação.

- struct _sockList *iniSockList : Ponteiro para uma lista de estruturas que armazena os clipboards que estão ligados ao clipboard local, que guarda um registo de todos os clipboards que estão ligados ao clipboard local. Esta lista é necessária para poder comunicar e fechar os sockets pelos quais são feitas as comunicações com esses clipboards quando o clipboard local é terminado, de modo a que os clipboards remotos possam sinalizar a sua terminação e consequentemente terminar a thread que estava a lidar com as comunicações com o clipboard que foi terminado.

- pthread_rwlock_t rwlock[10] : Vetor que armazena os *thread locks*. Cada um destes *locks* é usado para bloquear o acesso simultâneo entre *threads* a uma região diferente. Serão discutidas estes e outros métodos utilizados para o sincronismo entre regiões na secção 3 do relatório.

- int net_sock_fd : Este inteiro representa o *file descriptor* do socket associado ao *clipboard* ao qual o clipboard local é ligado de início. Este não consta da lista representada por *iniSockList*, visto que cada clipboard apenas se pode ligar a outro, e neste caso é útil separar este *file descriptor* dos restantes por razões de sincronismo entre clipboards que serão discutidas na secção 3. Caso o clipboard local não tenha sido ligado inicialmente a nenhum outro clipboard, este *file descriptor* terá o valor -1.

2.4. Fluxo de Tratamento de Pedidos

As aplicações podem fazer 3 tipos de pedidos a um servidor local. Pedir para ler a informação contida em qualquer uma das regiões do servidor (Paste), alterar a informação contida nessas regiões (Copy), ou ainda escolher uma região, e esperar que essa região seja alterada no clipboard. É discutido

de seguida o desenvolvimento e funcionamento de cada uma destas operações:

- **Copy:** No domínio da aplicação, esta operação corresponde à leitura de informação no clipboard, e é definida pela função *clipboard_copy() que consta da* **Library**. Esta função, a ser executada por uma aplicação após a verificação dos dados inseridos como argumento, prepara uma estrutura de dados de envio, mencionada na secção 2.1, colocando nesta o tamanho da mensagem, o tipo de operação a ser executada, a região na qual será feita a operação, e a mensagem em si. De seguida usa-se a função *memcpy* para transformar toda essa estrutura de envio num *bytestream* (tipo de dados adequado ao envio) que depois é enviado para o clipboard local.

De modo a processar o pedido pela aplicação, o clipboard recebe o *bytestream*, transforma-o de volta numa estrutura de envio, identifica a região e o tipo de pedido, neste caso **Copy**, e de seguida, caso o clipboard local seja a "raíz", ou seja, não tenha sido ligado a nenhum clipboard no momento em que foi iniciado, irá alterar a sua região com a informação recebida, e enviá-la a todos os clipboards a si ligados. Caso se tenha ligado inicialmente a outro clipboard, irá apenas enviar toda a estrutura de dados recebido para esse clipboard. Foi escolhido este tipo de troca de informações entre clipboards, devido ao problema de Sincronismo discutido na secção seguinte.

- **Paste:** Por sua vez, esta operação corresponde à alteração de informação numa região do clipboard, é definida pela função *clipboard_paste() que consta da* ***Library****.* Esta função, procede inicialmente como a operação anterior. No entanto, é alterado o tipo de operação especificado na estrutura de envio para **Paste**. Após a preparação da estrutura de dados de envio, é inicialmente enviada ao clipboard, uma estrutura de envio, com o objetivo de informar ao clipboard qual a região que a aplicação quer ler. Ao receber esta mensagem, o clipboard verifica na região indicada, qual o tamanho da informação nela contida, e procede então ao envio desse tamanho. Quando a função *clipboard_paste()* recebe esse tamanho, já pode alocar espaço para a mensagem a ser lida, e recebe seguidamente essa mensagem para o buffer alocado. Finalmente a aplicação copia esse buffer para um vetor *void* * que será obtido pela aplicação assim que a função termina.

- **Wait:** Esta operação é definida no domínio da aplicação pela função *clipboard_wait()* e consiste num pedido em que a aplicação fica parada à espera que a região passada como argumento na função seja alterada no clipboard. Esta função procede inicialmente como as já descritas acima, de seguida envia uma estrutura de envio com o pedido **Wait** de modo a informar relativamente à operação e à região. O clipboard por sua vez vai armazenar num buffer a informação que consta da região pretendida, e vai entrar num ciclo em que verifica constantemente se o que está na região é diferente do que armazenou nesse buffer. Assim que sai desse ciclo, o que significa que a região foi modificada, procede-se como na operação **Paste**. Inicialmente o clipboard envia o tamanho da informação da região atualizada, e de seguida

envia um buffer com essa informação, que é recebido pela função *clipboard_wait()* que já terá um buffer devidamente alocado para receber essa informação e a passar para a aplicação.

3. Sincronização

A partir do momento em que é decidido utilizar *threads* na programação deste sistema, há a necessidade desde logo de identificar quais os problemas de sincronização que vão surgir. Isto acontece porque as *threads* funcionam executando códigos diferentes paralelamente e alternadamente, pelo que se existirem variáveis que não sejam exclusivas de cada *thread*, o mais provável é termos problemas quando se tenta alterar a mesma variável, “ao mesmo tempo”, em duas *threads* diferentes, pois o resultado desta operação será indefinido, coisa que é muita má prática em programação.

Existe ainda um problema de sincronismo quanto à replicação entre servidores, pois o objetivo é que todos os servidores que estejam ligados em árvore tenham a mesma informação nas mesmas regiões. Isto origina um problema onde um nó pode receber duas atualizações diferentes ao mesmo tempo e, não tendo o problema do sincronismo resolvido, envia ambas as informações para os seus ramos adjacentes menos para o que lhe enviou a atualização, ou seja o servidor que enviou a primeira atualização recebeu a segunda, e o outro que enviou a segunda recebeu a primeira, ou seja, no fim deste processo vai haver um conjunto de servidores com a primeira atualização e outro conjunto com a segunda atualização, coisa que não é suposto. A resolução deste problema teve uma abordagem um pouco diferente, que não implicou a identificação de regiões críticas nem a implementação de exclusão mútua, pelo que iremos expor a solução no ponto 3.3.

3.1. Identificação das Regiões Críticas

Da forma como o sistema foi projetado, as únicas variáveis que podiam originar problemas de sincronismo são as variáveis globais. Como descrito em 2.3. foram usados três tipos de variáveis globais, não contando com as variáveis que nos garantem o sincronismo (*pthread_rwlock_t rwlock[10]*). No entanto só uma das variáveis globais é que levou a considerar regiões críticas.

É fácil de perceber que a variável global que despertou os problemas de sincronismo foi o vetor de regiões, pois pode acontecer haver várias aplicações e servidores a lerem e a escreverem ao mesmo tempo neste vetor.

Sendo assim, as nossas regiões críticas são todas aquelas onde há uma leitura ou uma escrita sobre este vetor de dez regiões (*struct _region regions[10]*).

3.2. Implementação de Exclusão Mútua

Como foi dito no ponto anterior, os nossos problemas de sincronismo devem-se a leituras e escritas, pelo que a escolha direta foi utilizar **Read-Write Locks**. Este tipo de *lock* permite que estejam a ocorrer várias leituras ao mesmo tempo, no entanto, quando há uma escrita não podem estar a haver leituras nem outras escritas ao mesmo tempo, e é isso mesmo que nos é grantido com os **Read-Write Locks**. Portanto, sempre que vamos alterar uma das regiões colocamos um **Write Lock**, sempre que vamos apenas buscar a informação que está numa determinada região utilizamos um **Read Lock**.

O facto da variável (*pthread_rwlock_t rwlock[10]*) ser um vetor deve-se a que os problemas só ocorrem se estivermos a fazer leituras e escritas na mesma região, ou seja, se temos dez regiões, vamos também ter dez **Read-Write Locks**, um para cada região.

3.3. Sincronismo entre Servidores

Para resolver este problema pensou-se na solução mais simples possível, visto que as soluções mais complexas iriam acrescentar componentes ao nosso sistema, algo que não era pretendido. Optou-se então por se perder um pouco mais de tempo de computação a atualizar todos os servidores, sempre que chegava nova informação a um deles, mas conseguiu-se simplificar bastante a solução para este problema de sincronismo.

A solução encontrada foi a seguinte: Sempre que um servidor recebe um pedido duma aplicação para guardar uma determinada informação, primeiro o servidor envia essa informação até ao topo da árvore. Só no topo é que a informação é guardada pela primeira vez e em seguida é enviada para todas as *folhas*. Como a informação veio do topo, as raízes já podem também guardar a informação nas suas regiões e continuar a enviar essa informação para baixo, até que todos os servidores da árvore a tenham recebido e guardado.

Esta solução garante que todos os servidores vão guardar a mesma informação que está no topo da árvore, ou seja, se se interromperem os pedidos de todas as aplicações e se esperar uns instantes, todos os servidores vão ter a mesma informação.

4. Gestão de Recursos e Tratamento de Erros

Neste projeto a gestão recursos foi feita de modo a otimizar o número de threads ativas e a computação que cada uma faz, visto que cada thread ocupa espaço e tempo de processamento, e tal é pouco ótimo a não ser que se esteja a realizar processamento útil. Deste modo apenas são chamadas threads adicionais no clipboard para lidar com novas ligações, algo necessário tendo em conta as funcionalidades do programa.

Quanto ao tratamento de erros, sempre que é alocada memória dinâmica, é feita a verificação de que ainda há memória disponível. Quando é utilizada qualquer função do domínio das *threads* e dos *locks* é sempre feita a verificação do valor de retorno das funções, que é -1 caso estas dêem erro. De modo a ser detetado o fecho de aplicações, é também verificado o valor de retorno das funções de *recv()* e *send()* que caso o socket para o qual é chamada a função se fecha, essa função retorna o valor 0.