TÉCNICO LISBOA

Systems Programming

Project Report

Distributed Clipboard

Professors: João Nuno e Silva
Ricardo Martins
2^{nd} Semester – 2017/2018
01/06/2018

Group 5:

81731 – Catarina Aleixo
81074 – Manuel Reis

I. Introduction

The main goal of this project is to develop a distributed clipboard on which any Application can copy and paste the content of a certain region, either running on the same or on different machines.

The Applications use a predefined API that was developed to copy/paste data to/from a clipboard.

II. Architecture

The system is composed of three components: an API, a Library and a local clipboard.

The API is used to allow programmers to develop applications that use the distributed clipboard.

The library is an implementation of the API and includes the code needed to interact with the clipboard.

The local clipboard handle both connections from local applications and remote clipboards, i.e, that are running on different machines. The local applications can copy and paste data from the local clipboard. The remote clipboards replicate data between them when there's a change in one of its ten regions.

The workflow of the developed program is represented in figure 1 and it's explained in the following sections:

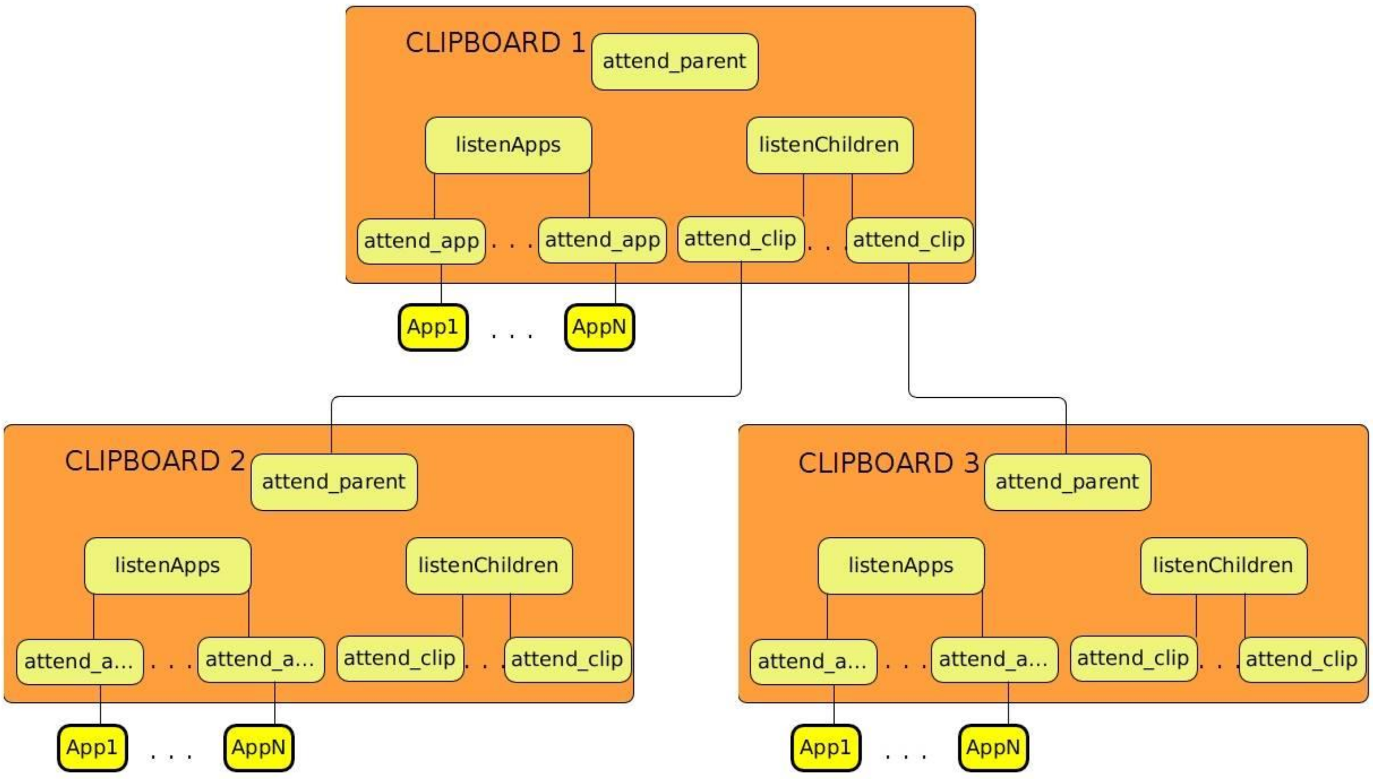

Figure 1 - Workflow

A copy instruction is performed when an application wants to change a certain region in its local clipboard. Every time a copy is invoked, the local clipboard has to send the region changed to the other clipboards, in order to keep all clipboards synchronized.

The paste instruction sends the content of a given region to the application.

The wait instruction consists of waiting that a certain region change its content.

When a remote clipboard it's connecting to another one, first it has to synchronize all its regions. The termination of this process it's recognize by the clipboard that is connecting when it receives an S instruction.

The communication between remote clipboards it's done using stream sockets. On the other hand, the communication between applications and its local clipboard is done using UNIX sockets.

II.II Order Processing Flow

Before explaining how each instruction is processed, it's important to clarify how a clipboard can handle many applications and other remote clipboards, simultaneously.

Each clipboard starts by checking if there is an available clipboard on which it will try to connect. If there's no other clipboard or if the connection wasn't properly established, means that the clipboard will be on single mode. Afterwards, it will create a thread invoking the *pthread_create()* function that will be dedicated to accept possible future connections of other clipboards. Every time a new connection is made another thread is created, and it will handle the communication between the local clipboard and the remote clipboard that was connected, calling the *attend_clip()* function.

Thereafter, the main thread will be accepting connections of the applications and, every time a connection to an application is done, a new thread is created, and it will be dedicated to handle requests from that application calling the *attend_app()* function.

When a certain request comes from an application or from a remote clipboard, then the *handleRequest()* function is called. In this function, the request is analyzed, starting by checking if it fits the protocol, otherwise it's an unknown instruction, meaning that the local clipboard will remain intact.

- **Copy:**

Each application can modify the regions of the clipboard to which is connected by sending a copy instruction to the clipboard. Every time a copy is done, there's a need of synchronizing all the other clipboards, by sending the region that was modified.

A copy is done to a certain region that has to be between 0 and 9, since there are 10 regions. This instruction implies changes in memory that is shared by all threads that, either handle remote clipboards or applications, which means that there's a critical region during this changing process. Every time an application or a remote clipboard wants to make a copy, it has to lock the corresponding region. In order to guarantee exclusive access to the regions when a copy it's invoked, there is a *pthread_rwlock_t* for each clipboard region. The write lock is done using the *pthread_rwlock_wrlock(*) function. The unlock of the region it's only performed after changing its content.

Afterwards, a broadcast is done to all the other clipboards to keep them synchronized.

After changing the region and before unlocking it, a *pthread_cond_broadcast()* is done releasing all the threads, that are waiting for a change in that region, on the *pthread_cond_wait()* function, on *waitRegion()* function.

- **Paste:**

When an application invokes a paste instruction, the thread that handles the communication between the local clipboard and the application has to send the content of the region that was requested. In order to guarantee that no one is changing the content of that region when the paste is being done, a read lock is done using the *pthread_rwlock_rdlock()* function. The *pthread_rwlock_t* variables are the same that are used when a copy is done.

For the purpose of decreasing the critical region, the content of the requested region is copied to a local buffer, which will decrease the overhead of the other clipboards of invoking a paste over the same region, since the unlock of the region is performed before sending its content.

- **Wait:**

The wait instruction it's performed when an application wants to wait until the content of a specified region changes. When that region changes, the application will be notified as a paste had been invoked, meaning that will receive the most recent content by that time.

When the threads that are waiting for a certain region received the desired content, implies that there was a copy on that region. Any thread that handles a wait instruction will be redirected to the *waitRegion()* function and will acquire a mutex of that region and then it will release that lock, since it will wait on the *pthread_cond_wait()* function, that receives as argument a *pthread_cond_t* variable and a mutex that are dedicated to that region, meaning that there is not only a mutex, but also a condition variable for each region, in order to keep the waiting on different regions, a parallel process, although the threads waiting for the same region will perform sequentially that critical region.

Each thread waiting for the same region will be released as soon as the *pthread_cond_broadcast()* function is performed, after the copy instruction changes the content of the region. After that, the mutex of the region will be sequentially acquired by each thread that is waiting on the same region, to unlock the mutex.

III. Synchronization

In order to keep track of all the remote clipboards that are connected to a local clipboard, each local clipboard holds a list of a structures called *Elm* that contains the file descriptors of the clipboards that are connected to it, and a mutex (*pthread_mutex_t* variable), of each remote clipboard.

The first synchronization of each clipboard happens before any thread is created, calling the *connect2parent()* function. This function returns the file descriptor that allows the communication between clipboards, if there's already a clipboard running and if the connection is properly established. Otherwise, it will return -1, which indicates that the clipboard will be running on single mode. If the value returned corresponds to a valid file descriptor, then the *initialSync()* function is called and the remote clipboard will be waiting for receiving each non empty region, according to the communication protocol described above, knowing that when receives an S, the synchronization ended. Afterwards, a thread will be created to receive and process each request that could come from the remote clipboard.

Having in mind that the main thread of each program will be dedicated to accept new connections from applications, before doing it, it has to create a thread that will be accepting possible future connections from other remote clipboards, calling the *listenChildren()* function.

The first step when entering the *listenChildren()* function is to create a socket stream that will be listening new connections, calling the *setupParentListener()* function. Then the thread will be accepting new connections from other remote clipboards. When a new connection is established, a new thread will be created and will call the *attend_clip()* function and a new *Elm* node will be added to the list. Any time that a thread needs to add or check something on the list, it has the lock the access to it. However, to make possible that two or more threads can access the list without having to wait until one thread reaches the last node, each thread when enters the list acquire the mutex of the first node and only unlocks it after acquiring the mutex of the next node. This procedure is done because, for example, if two messages are copied to the same region, the thread that acquires first the mutex of the first node, it will always be the one that will reach the last node in the first place, which guarantees that the order of the messages will be the same in each remote clipboard, meaning that they all will be well synchronized.

Between clipboards the messages that will be exchanged are only copies, since every time an application makes a copy, the clipboard has to broadcast the region changed to the other clipboards. This process is made establishing that the clipboard that it's working on single mode it's the one that will order the changing. This turns the synchronization problem easy to solve since, when an application changes a region, it sends it to its parent clipboard (the one to which it's connected) and the one that receives it do the same without changing their regions yet, which means that only the first clipboard that it's not connected with no other clipboard won't send it. This message is sent in the *broadcastReq()* function where first, the thread will check if its parent is still running, *i.e*, if the clipboard isn't running on single mode.

This checking part has to be done locking the shared variable *parent_fd* with a mutex called *parentMutex*, since another thread could be communicating with the parent file descriptor and if it can't reach its parent, it will change the file descriptor to -1, which means that the clipboard will run on single mode from that moment on.

When the message reaches the top clipboard, the corresponding region will be modified, and the clipboard will notify all the remote clipboards that are connected to it and they will do the same until the message reaches the last clipboard that it hasn't anyone connected to it. Except the top clipboard, all the others only change its region when receives a message from its parent clipboard. This guarantees that everyone will be synchronized. All the process of changing the region is equal to the process of a copy that is done by an application and it's already described in the order processing flow section.

It's also important to refer that, each time a remote clipboard disconnects from its parent, the corresponding node of the list its updated changing the value of the file descriptor, but the node isn't removed from the list. Instead, the value of the file descriptor is changed to -2 and, when other remote clipboard establishes a new connection, it's called the *reuseNode()* function which checks the file descriptor and reuses a node if that value is -2. Otherwise, adds a new node to the list.

III.I Identification of the critical regions

The critical regions in the synchronization are when a thread wants to access the list of remote clipboards that are connected to it and when a thread wants to change a region when receives a copy instruction from its parent clipboard.

III.II Implementation of mutual exclusion

The mutual exclusion it's implemented using mutexes to lock the nodes of the list and in the copy instruction when changing a region, which it's already described in the order processing flow section.

IV. Resource Management and Error Handling

The resource management and the error handling are taking into account in the code implementation since all memory is freed, including both lists, one that holds the file descriptors of the applications and another one that contains the file descriptors of the remote clipboards, the detach of the threads and the ten regions of the local clipboard. There are signals that are also handled, such as the SIGPIPE and Ctrl-C.