

***Clipboard* Partilhado**

Licenciatura Engenharia Informática e de Computadores

Projeto e Seminário

Semestre Verão 2016/2017

Desenvolvido por:

André Carvalho

Orientador:

Pedro Pereira

Lisboa, setembro 2017



***Clipboard* Partilhado**

Licenciatura Engenharia Informática e de Computadores

Projeto e Seminário

Semestre Verão 2016/2017

Desenvolvido por:

André Carvalho nº 41839

|  |
| --- |
|  |

Orientador:

Pedro Pereira

|  |
| --- |
|  |

**Índice**

**Introdução** ……………...…………………………………………………………………………..……... 6

**Resumo** ……………………………………………………………………………………………..…....... 7

**Capítulo I – Divisão das componentes** …………………………………………………………………….. 8

**Capítulo II – Servidor** ………………………………………………………………………………………….. 9

**Escolha de base de dados** ………………………………………………………………………..… 9

**Criação de contas de utilizador** ……………………………………………………………... 10 - 11

**Renderização da página *Home*** ……………………………………………………………….…... 12

***API* disponível** ………………………………………………………………………….................... 13

***Formato de resposta*** …......................……………………………………………….................... 14

**Alterações do *clipboard*** …..........................……………….………………………............. 15 - 17

**Sincronização**…..........................……………….………………………..................................... 18

**Requisitos não funcionais**…….…...............................….………………………..................... 19

**Capítulo III – Aplicação *Android*** …………………………………………………………………………... 20

**Interação com o servidor** ………………………………………………………………………….. 21

**Componentes** ….....................................……………………………….……………………. 22 - 23

**Deteção de alterações no *clipboard*** ………………………………….…………………………. 24

**Deteção de alterações no *clipboard* global (lado do servidor)** ….……………….….... 25 - 26

**Histórico** …………………………….…………………………………………………….…............. 27

**Limitações** …………………………….…………………………………………………….….......... 28

**Capítulo IV – Aplicação *Windows*** …………………………………………………………………………. 29

**Interação com o servidor** …………………………………………………………………….. 30 – 32

**Componentes** ….....................................……………………………….……………………. 33 - 34

**Deteção de alterações no *clipboard*** …………………………………………………………...... 35

**Deteção de alterações no *clipboard* global (lado do servidor)** ………………….…............. 36

**Histórico** ………………………………………………………….…............................................... 37

**Limitações** …………………………….…………………………………………………….….......... 38

**Conclusão** …………………………………………………………………………....................................... 39

**Índice de figuras**

**Figura 1 - Esquema do sistema** ……………………………………………………………….……….. 7

**Figura 2 – Modelo EA**….........................…….………………..…………………….......………………..… 9

**Figura 3 - Processo de Autenticação**……….………………..…………………….......………………… 11

**Figura 4 – Comunicação entre serviços** …................................................…….......………………… 23

**Figura 5 - Primeira abordagem para obtenção de dados do servidor** ……….......………………… 26

**Figura 6 - Comunicação utilizando *websockets*** …...................................…….......………………… 31

**Figura 7 – Comunicação entre serviços** ………….............………………..…….......……….………… 34

**Índice de tabelas**

**Tabela 1 – Comparação entre *WebSockets* e *API*** ………………………...……………….……… 15

**Tabela 2 – Tabela de comparação entre *Volley* e *Retrofit***  ……..……...……………….……….. 21

**Introdução**

No mundo conectado de hoje em dia, apesar de existirem múltiplos dispositivos ligados entre si através da internet, existem algumas funcionalidades que ainda não se encontram integradas, como o caso do *clipboard* dos diferentes dispositivos. Em vez de termos um único *clipboard* partilhado pelos múltiplos dispositivos/sistemas operativos temos vários onde cada um pode conter informação diferente.

Este projeto tem como objetivo o desenvolvimento de um serviço, que visa unificar o *clipboard* dos diferentes sistemas operativos num só.

Um exemplo de utilização é um utilizador visualizar um contato telefónico no seu *laptop*, realizar a ação de *Copy*, e em momentos esse mesmo contato telefónico copiado, estar pronto a ser usado no seu telemóvel, através da ação *Paste*.

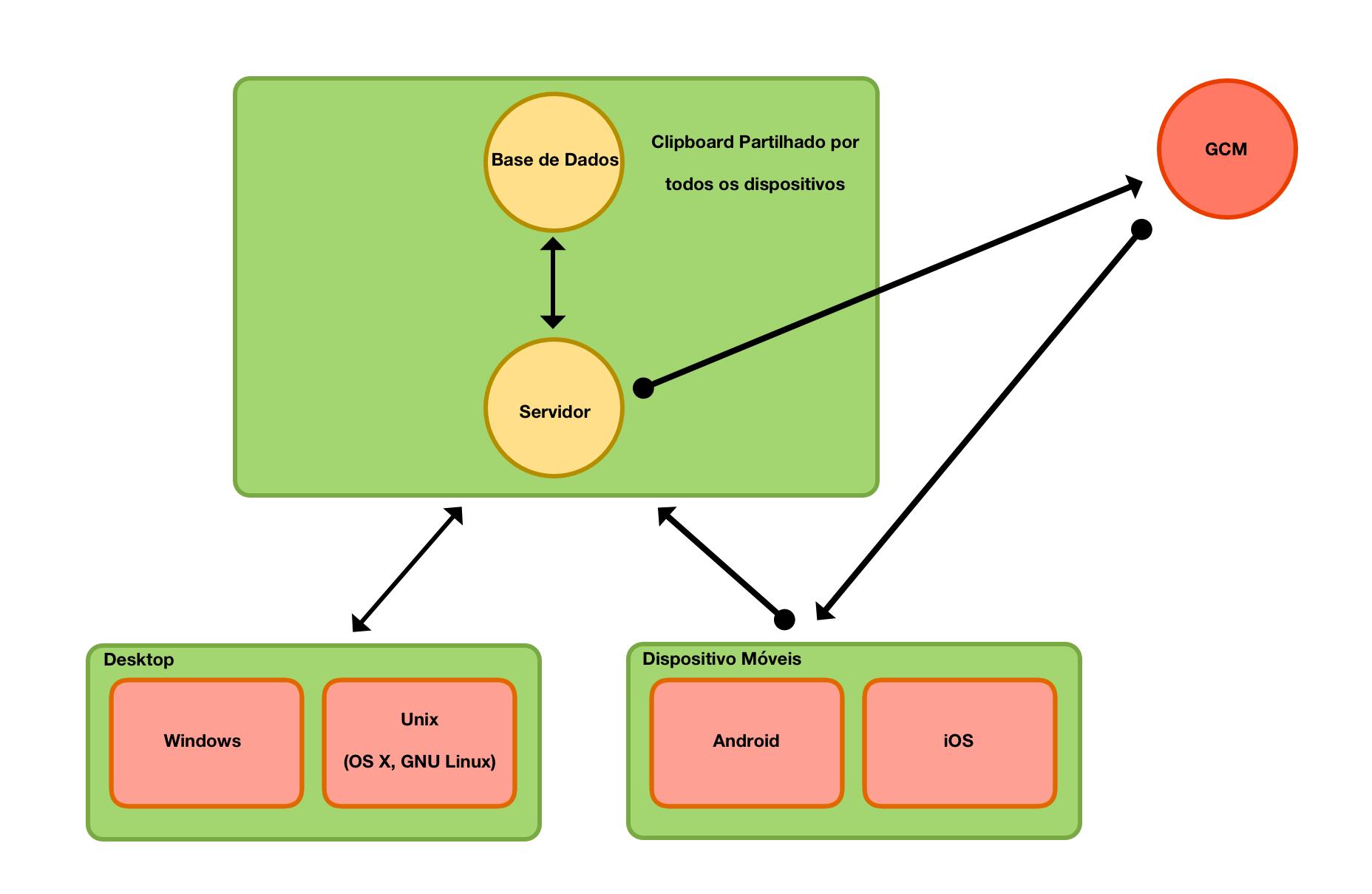
**Resumo**

O sistema implementado no decorrer deste trabalho pode ser resumido à Figura 1 – Esquema do sistema.

Através da figura podemos observar que o servidor disponibiliza uma *API* (descrita em mais detalhe no capítulo do servidor) que permite a troca de conteúdo entre os diversos dispositivos que podem ter diferentes sistemas operativos.

Este projeto limita-se a desenvolver uma aplicação para *Windows* e outra para *Android*.

De notar que existem uma separação entre as ambas as aplicações.

Enquanto que a aplicação *Android* recorrem à *API* fornecida para enviar conteúdo e ao serviço de *GCM* disponibilizado pela *Google* para a receção do mesmo (devido aos múltiplos cuidados a ter num desenvolvimento de uma aplicação móvel), a aplicação *Windows* como não tem mecanismos semelhantes ao de *GCM* o servidor permite a comunicação através de *WebSockets*. Assim, a aplicação *Windows* não só passará a obter novo conteúdo de uma forma passiva, visto que é o servidor que fica responsável por enviar face à uma solução onde o cliente através de *poll* ficaria à espera de novo conteúdo, este também permite o envio de novo conteúdo através do mesmo canal. Sendo assim o cliente *Windows* não necessita de recorrer a *API* para gerir o conteúdo.

**Figura 1 - Esquema do sistema**

**Capítulo I – Divisão das componentes**

Este projeto poderia seguir, pelo menos, duas implementações:

1. Uma implementação onde os dispositivos comunicariam entre si através de tecnologias como o *bluetooth* ou o *wifi*. Esta abordagem teria como desvantagens, por exemplo a obrigatoriedade de todos os dispositivos estarem em proximidade e conectados entre si, e poderia apresentar limitações no número máximo de dispositivos que o utilizador poderia usar.
2. Outra abordagem seria a de desenvolver uma aplicação servidora que guardaria o estado mais recente do *clipboard*, e esta disponibilizaria uma *API* que permitisse realizar a comunicação entre os diferentes componentes deste projeto.

Foi escolhida a segunda abordagem, visto que apresenta um maior número de vantagens em relação à primeira, nomeadamente a possibilidade de suportar um maior número de dispositivos.

Assim, este projeto teria como obrigatoriedade o desenvolvimento de uma aplicação servidora, contudo ficaria por definir as componentes que iriam ser suportadas para fins demonstrativos.

Optou-se por realizar uma componente *desktop* e uma componente móvel.

A nível da componente móvel a escolha foi bastante fácil visto que de entre os diversos sistemas *(iOS, Android, Windows Phone*), o *Android* é o único que tem uma vasta visibilidade ao nível dos utilizadores, e não apresenta restrições ao nível do sistema como o caso do *iOS*.

A nível da componente *desktop* inicialmente optou-se por desenvolver para *macOS* que existia algum interesse na aprendizagem linguagem de programação *Swift*.

Contudo, devido a algumas restrições na falta de uma *API* concreta para tratamento de eventos do *clipboard*, optou-se pela realização de uma componente para o sistema *Windows*.

Assim, este projeto será composto pelas seguintes componentes:

* Servidor
* Aplicação *Android*
* Aplicação *Windows*

**Capítulo II - Servidor**

Em termos de tecnologias, para o desenvolvimento do servidor ponderou-se a utilização de *JEE, ASP.net* e da *framework* *Spring*.

Uma vez que a *framework* *Spring* estava a ser lecionada na cadeira de Desenvolvimento de Aplicações Web, optou-se por usá-la.

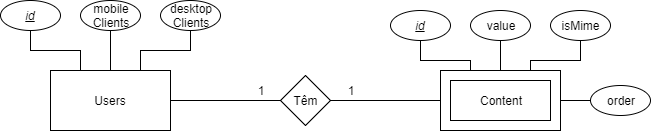
**Base de dados**

Para a realização deste projeto seria possível recorrer a uma base de dados relacional como uma não relacional.

Optou-se pela utilização de uma base de dados não relacional (*MongoDB*) pelos seguintes motivos:

* Iniciar a aprendizagem de base de dados não relacionais
* Uma maior facilidade em escalar a aplicação, caso seja necessário
* Um maior suporte pelos diversos *PaaS*, onde a aplicação servidora seria colocada.

Para este projeto a base de dados teria que dar suporte a entidades de utilizadores, o conteúdo destes e dos seus dispositivos. Assim, a base de dados tem o modelo *EA* ilustrado pela figura 2. Existiria duas tabelas ambas identificadas por um *id* único a cada utilizador. A tabela *Users* serviria para identificar um utilizador e os seus repectivos dispositivos guardados nos campos *mobileClients* e *desktopClients*. A tabela *Content* conterá um *id* para identificar o utilizador, um campo *value* para guardar o último valor copiado, um campo *isMime* a indicar se o value se trata de texto ou imagem, e um campo order para indicar a ordem do último pedido.



**Figura 2 – Modelo EA**

**Contas de utilizador**

O servidor suporta uma gestão de contas de utilizador recorrendo ao protocolo *OAuth* e ao serviço de *OAuth* disponibilizado pela *Google*.

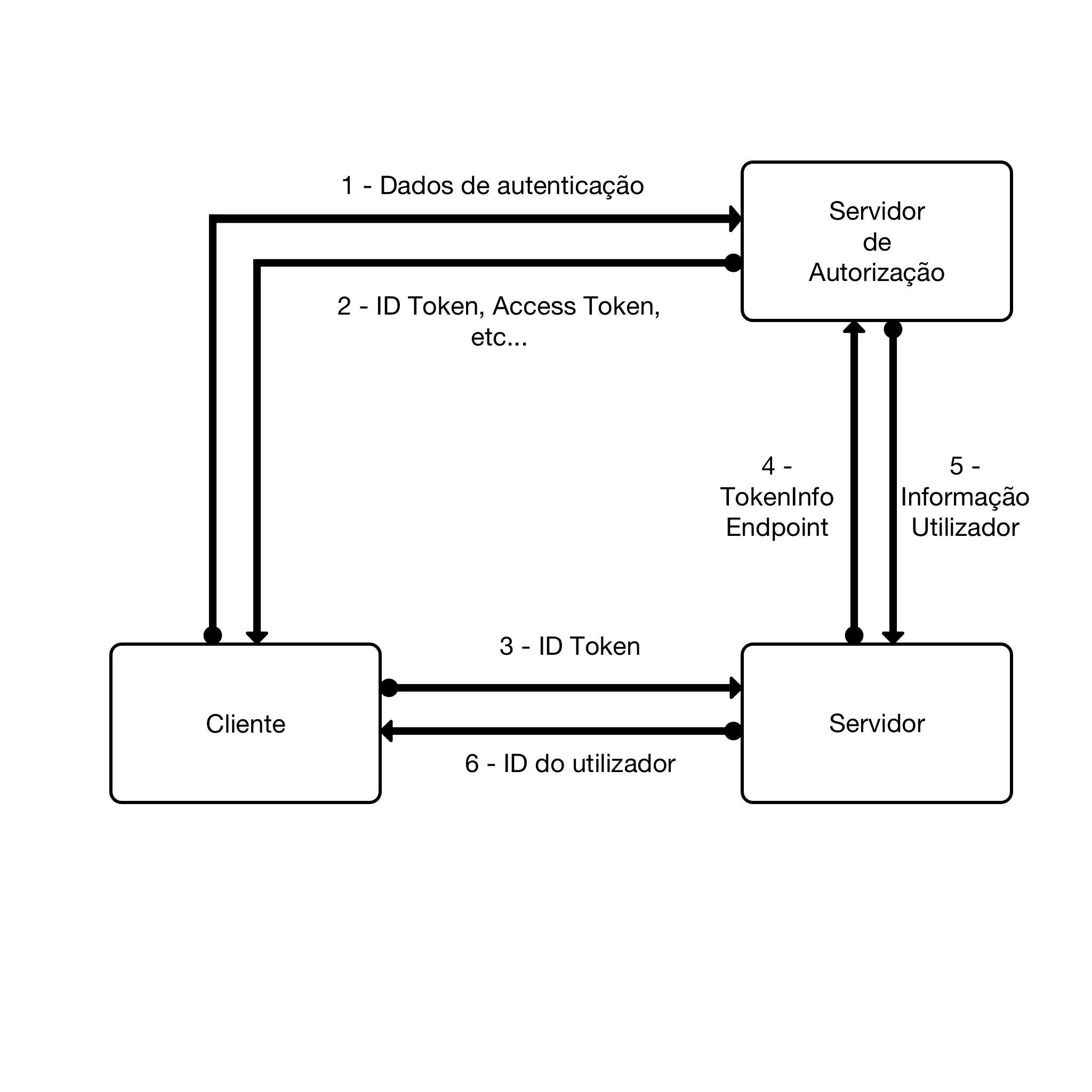
Assim, o processo de autenticação de um utilizador será diferente aos que estamos acostumados, onde a aplicação pede um *email* e uma palavra-passe e envia ambos para o servidor e o servidor trata de verificar esses dados com a sua base de dados.

Recorrendo ao protocolo de *Oauth*, em vez de existirem apenas duas entidades, cliente e servidor, passaram a existir três, o cliente, servidor e servidor de autorização.

Assim, um processo típico de autenticação seguindo o protocolo de *OAuth* implica:

* Comunicação do cliente com o servidor de autorização de forma a que este possa realizar a autenticação. No fim do processo de autenticação o cliente recebe um *ID Token*, um *Access Token*, etc…
* Uma vez que o cliente tenha na posse esta informação poderá comunicar ao servidor isso. Este realiza um pedido para */api/authenticate* onde envia como parâmetro de *header Authorization* o valor do *ID Token* recebido.
* O servidor ao receber este pedido valida com o servidor de autorização o *ID Token* recebido. Caso se trate de um *ID Token* válido o servidor de autorização devolverá como resposta ao servidor informação relativa a conta de utilizador como por exemplo o ***sub*** do utilizador (semelhante à um *id* único do utilizador), que será utilizado para identificarmos os utilizadores.
* Uma vez identificado o estado da conta do cliente, essa informação é comunicada a aplicação cliente.

Assim, o processo de autenticação poderá ser descrito segundo a figura 3.



**Figura 3 – Processo de Autenticação**

**Página *Home***

O servidor não só tem o papel de disponibilizar a *API*, como também tem o papel de consumidor da mesma.  
 Caso o utilizador visite a página raiz do servidor ([www.projecto1617.herokuapp.com](http://www.projecto1617.herokuapp.com)), este devolverá uma página que conterá como funcionalidades a realização de uma inscrição de uma conta de utilizador, sua gestão, e ainda a possibilidade de realizar a descarga da aplicação para o sistema operativo ao qual o utilizador utilizou para visitar a página, caso este esteja disponível.  
 Para a realização do preenchimento de informação referente ao sistema operativo no qual o utilizador utilizou para visitar o site de inscrição/descarga do *software*, não é aconselhável usarmos *html* estático uma vez que à informação apresentada varia de sistema para sistema. Assim, recorreu-se a uma *view engine*.  
 Optou-se por usar o *Handlebars*, visto que é um *view engine* popular e esta permite operações de controlo de fluxo, ciclos, etc.

***API* disponível**

O servidor disponibiliza a seguinte *API REST*:

**• *POST /api/account***

**• *PUT /api/account***

**• *PUT /api/registerDevice***

**• *GET /api/socket***

**• *PUT /api/push***

**• *POST /api/push***

Uma descrição de cada método da *API* poderá ser consultada no ficheiro “*API”*, disponível em anexo.

De notar que a *API* não disponibiliza um *endpoint* que permita a obtenção de informação.

Isto deve-se ao facto de o mesmo disponibilizar maneiras mais eficazes (as quais veremos a seguir) de forma a que o cliente tenha sempre o conteúdo mais recente.

De notar também que se optou por devolver o *id* do utilizador codificado quando este se autêntica com sucesso e utilizar esse *id* para as operações de gestão do conteúdo simplesmente para facilitar o *debug* das operações. Num cenário de produção o mais correto seria continuar a utilizar o *id token* fornecido pelos servidores de autorização da *Google* como forma de identificar o utilizador.

**Formato da resposta**

Uma vez que os consumidores desta *API* são aplicações com interfaces gráficas, para facilitar a notificação do utilizador em caso de erro, a *API* disponibiliza como resposta de os todos os pedidos uma mensagem *JSON* com a descrição do resultado da ação realizada no seguinte formato:

* ***Title***: um breve *headline* do resultado da ação realizada.
* ***Description***: uma descrição do resultado da ação realizada.

Através destes dois campos, descritos numa forma não técnica, seria possível alterar as funcionalidades do servidor, sem necessariamente alterar as respostas à erros das aplicações cliente.

Contudo, nem todos os pedidos são iguais. Alguns pedidos é necessário a transmissão de dados extra. Caso seja este o caso esses dados serão colocados no objeto ***data*** da resposta.

De forma a facilitar a deteção de erros aquando o *debug* das aplicações cliente, estas poderão guardar em *log* o campo ***error*** caso o pedido realizado a *API* tenha resultado num erro.

**Alterações do *clipboard***

A distribuição de novo conteúdo entre dispositivos varia caso estes sejam dispositivos móveis ou dispositivos *desktop*, contudo, ambas as soluções têm algo em comum, não obrigam a que a aplicação cliente tenha o trabalho de obter a nova informação, deixando isso a cargo do servido. Não só esta solução traz simplicidade para a aplicação cliente, como permite adicionar funcionalidades que não poderiam ser facilmente realizadas através de uma *API web*, como o caso de um histórico do *clipboard* de cada dispositivo, ou a possibilidade de remoção de um dispositivo da conta de utilizador. Uma comparação entre os dois métodos de comunicação com o servidor pode ser descrita pela tabela 1.

|  |  |  |
| --- | --- | --- |
|  | **Vantagens** | **Desvantagens** |
| ***API*** | ***-*** Uma maior facilidade de utilização (?) | ***-*** O cliente necessita de se preocupar com a obtenção de novos dados. Isto pode ser feito através de *poll* (Elevado custo pois estará sempre a abrir uma nova comunicação). |
| ***WebSockets*** | ***-*** O cliente não necessita de se preocupar com a obtenção de novos dados uma vez que estes são enviados automaticamente para ele.  **-** Simplifica a implementação da funcionalidade de remoção de dispositivos. | ***-*** O cliente tem que realizar *pings* no canal, caso contrário a conexão ficaria fechada. |

**Tabela 1. Comparação entre *WebSockets* e *API***

Assim, para a aplicação *Android* optamos pela utilização do servido da *Google,* o *GCM*, e para a aplicação *Windows* recorremos à *WebSockets*.

Em seguida descrevemos as ações necessárias para a notificação recorrendo a ambos os serviços.

* **GCM**

Quando pretendemos notificar os dispositivos registados de alguma alteração ao estado do *clipboard*, o servidor realiza um pedido ***post*** para o *endpoint* disponibilizado pela *Google* (<https://gcm-http.googleapis.com/gcm/send>), onde o corpo do pedido conterá uma mensagem com o seguinte formato *JSON*:

{

“*to*”: *deviceID*,

“*data*”: {

“*action*”: “*store*”,

“*content*”: mensagem…,

“*isMIME*”: *true/false,*

“*order*”: *n,*

}

}

Os campos ***content*** e ***isMIME***, conterão texto e o valor *false* caso o último elementocopiado seja texto ou um ***link*** e ***true*** caso último elementocopiado seja uma imagem.

* ***WebSockets***

Uma vez que não existe o serviço de *GCM* para dispositivos *desktop, por exemplo*, o servidor implementa uma alternativa através de *WebSockets.*

Aquando o servidor pretenda notificar um dispositivo que utilize para comunicação os *WebSockets* este envia através da comunicação já estabelecida (O processo de estabelecimento de comunicação através de *WebSockets* será visto no capítulo IV – Aplicação *Windows*) a seguinte mensagem *JSON*:

{

“*action*”: “*store*”,

“*content*”: texto…,

“*isMIME*”: *false,*

“*order*”: *n,*

}

Caso pretenda notificar que o último elemento copiado foi texto (da mesma forma que notifica os dispositivos móveis).

Para a notificação de pedidos multimédia uma vez que não existem as limitações de tamanho da mensagem impostas pelo serviço *GCM*, é possível enviar o ficheiro diretamente. Neste caso o corpo da mensagem enviada será:

{

“*action*”: “*store*”

“*content*”: ficheiro,

“*filename*”: nome do ficheiro,

“*isMIME*”: *true,*

“*order*”: *n,*

}

**Sincronização**

Tratando-se de um projeto que implementa um sistema distribuído poderia ocorrer dessincronização do elemento que foi copiado mais recentemente.

Um exemplo deste problema é quando um dispositivo do utilizador realizasse a copia de texto e enquanto este comunicasse com o servidor essa informação, o mesmo recebesse uma notificação com novo conteúdo para o *clipboard*.

Para a resolução deste problema o servidor implementa a identificação de cada pedido *push* com um *id* único por cada conta de utilizador.

Assim, é possível garantir uma ordem temporal do valor do *clipboard*.

Do lado das aplicações cliente, estas uma vez implementam uma *queue* de pedidos em execução e como guardam o último número de ordem recebido por elas de uma forma simples é possível determinar se a notificação recebida seria a mais recente através da seguinte expressão matemática:

Uma vez determinado o estado da notificação, caso esta se trate do elemento mais recente, procedemos a alteração do *clipboard* local com o valor recebido e ainda guardar esse valor no histórico do dispositivo.

De notar que nas implementações tanto da aplicação *Android* como da aplicação *Windows*, existe um problema ao alterar o *clipboard* local do dispositivo onde não é possível notificar o sistema que o texto não foi copiado por ação do utilizado, provocando assim problemas onde o texto recebido acabaria por ser reenviado para o servidor caso este tenha sido considerado como o mais recente.

Para a resolução deste problema, aquando a verificação que o elemento recebido se trata do elemento mais recente, adicionamos o valor a uma lista onde a cada notificação de uma ação de *copy* iremos proceder à verificação do valor copiado. Caso este se encontre na lista, podemos assumir que se trata de um valor que recebemos numa notificação passada podendo assim garantir que não é necessário realizar o envio desse valor para o servidor. Uma vez verificado e caso este valor esteja presente na lista, será removido de forma a garantir o normal funcionamento da aplicação.

**Requisitos não funcionais**

Durante o desenvolvimento do projeto e devido as diversas soluções tomadas, surgiram requisitos dos quais se ponderou que faria sentido implementar.

Esses requesitos são a opção de remover um dispositivo da conta de utilizador e um histórico dos elementos copiados.

* **Remoção de um dispositivo**

Caso o utilizador não pretenda que um dos seus dispositivos comunique com o serviço este poderá remove-lo através do painel de conta disponível em [www.projecto1617.herokuapp.com](http://www.projecto1617.herokuapp.com). Ao desassociar um dispositivo da sua conta este receberá uma mensagem através do serviço de *GCM* ou *WebSockets* com o seguinte corpo:

{

“*action*”: “*remove*”

}

Ao receber a mensagem este procederá a destruição de todos os dados guardados no dispositivo.

* **Histórico**

Inicialmente, ponderou-se que o servidor guardasse todos os elementos que este recebesse, contudo, isto poderia não ser uma solução viável, pois assim a base de dados poderia ficar sobrecarregada prejudicando assim a sua *performance*. Este problema poderia ser aliviado recorrendo a um *cluster* de base de dados, contudo, pensou-se que uma vez que a notificação de novo conteúdo é realizada de forma passiva, isto é, o trabalho estaria ao cargo do servidor existiria assim, uma garantia que mesmo que o dispositivo não estivesse ligado este acabaria por receber o conteúdo, graças a uma *cache* implementada pelo *GCM* ou por uma *cache* implementada pelo servidor para as aplicações que recorrem a *WebSockets*.

Optou-se por deixar um histórico de conta e passarmos a um histórico de dispositivo, onde cada dispositivo teria um histórico desde o momento da sua primeira utilização. Desta maneira conseguimos remover os possíveis *bottlenecks* que seriam introduzidos pela primeira solução.

**Capítulo III - Aplicação *Android***

A componente *Android* do projeto é desenvolvida recorrendo a linguagem de programação *Java*.  
 No momento do início do desenvolvimento desta componente, as linguagens oficialmente suportadas eram o *Java* e *o C/C++*. Visto que não existia necessidade de funcionalidades de baixo nível optou-se por realizar esta componente recorrendo ao *Java* 1.7. Mais recentemente começou-se a adotar a linguagem *Java* 1.8, para tirar partido de interfaces funcionais, *streams*, etc.

**Interação com o servidor**

Para a comunicação entre o servidor e a aplicação *Android* esta recorre à *API REST* referida anteriormente. Para a realização de pedidos *HTTP*, no planeamento foram analisadas duas bibliotecas a *Volley* mantida pela *Google* e o *Retrofit* mantida pela *FourSquare*.

Na tabela 2 está a comparação entre ambas para justificar a escolha tomada.

|  |  |  |
| --- | --- | --- |
|  | **Vantagens** | **Desvantagens** |
| ***Volley*** | ***-*** Dá suporte a um grande número de funcionalidades que não são suportadas pelo *Retrofit*. | ***-*** Acréscimo de complexidade face a biblioteca *Retrofit* |
| ***Retrofit*** | ***-*** Documentação.  **-** Parse automático de JSON.  ***-*** Simplicidade. | ***-*** Não suporta a geração de pedidos para *URLs* desconhecidos em tempo de compilação |

**Tabela 2 - Tabela de comparação entre *Volley* e *Retrofit***

Optou-se, por recorrer ao *Retrofit* devido ao parse automático de *JSON* e a sua simplicidade de utilização, para além de que a biblioteca *Volley* continha muitas funcionalidades não necessárias para este trabalho. Contudo, caso no futuro se pretenda implementar uma distribuição de trabalho entre múltiplos servidores ter-se-á que ponderar se não valerá a pena alterar para a biblioteca *Volley* visto que esta permite a criação de *URLs* em tempo de execução.

**Componentes**

No decorrer do desenvolvimento da aplicação *Android* foram utilizados diversos componentes. Este subcapítulo serve para os identificar e explicar o motivo da sua utilização.

***Activities***

A aplicação *Android* é composta por três *activities*. Uma *activity* para realizar o *login*, outra para ajustar as definições do serviço/aceder ao histórico do dispositivo e ainda uma *activity* de “acerca” para o detalhe de licenças das bibliotecas utilizadas.

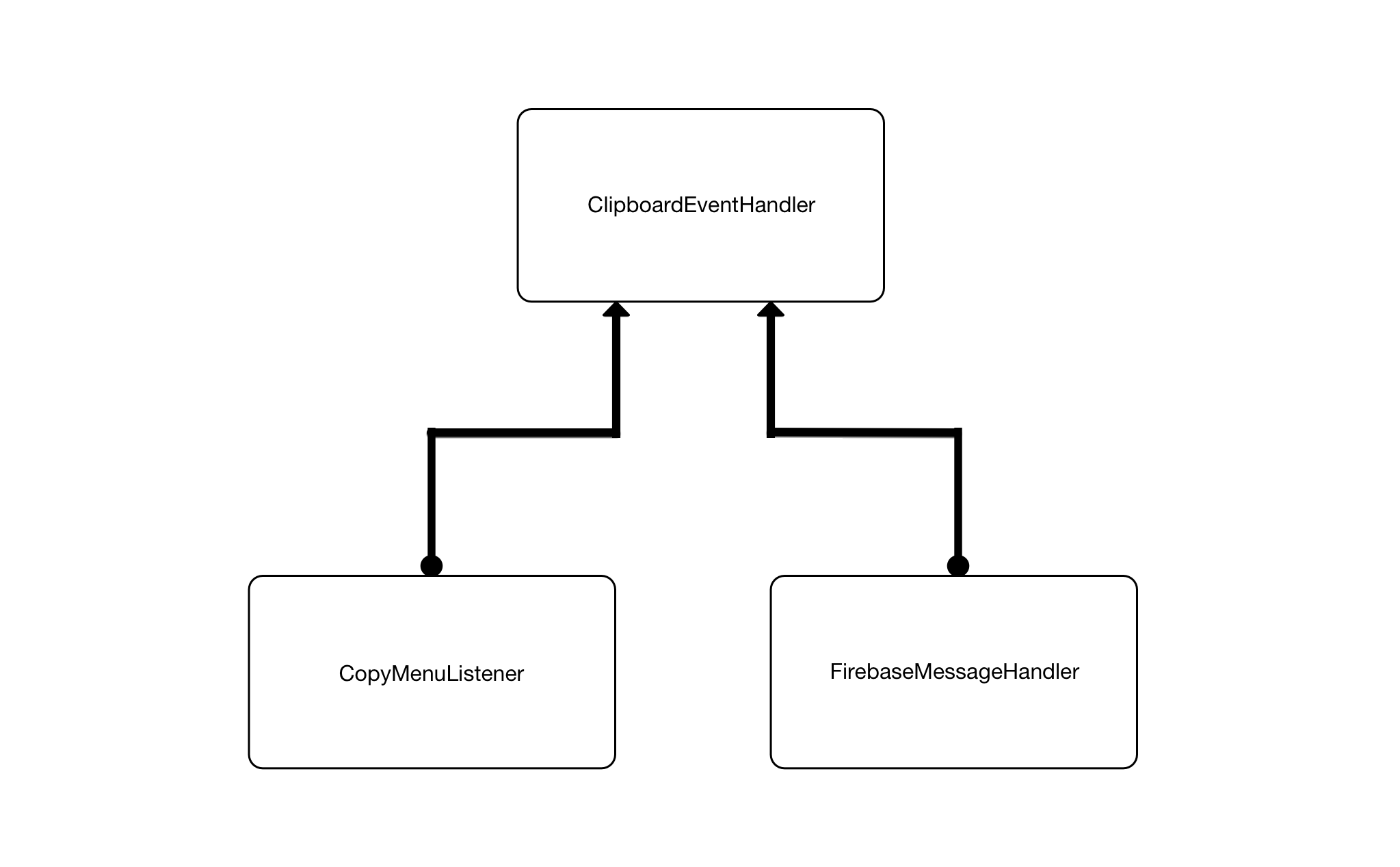
***Serviços***

A parte mais fundamental de todo o código da aplicação *Android.*

Esta aplicação é composta por três serviços.

* ***CopyMenuListener***: Serviço responsável pela adição de um listener ao clipboard local do dispositivo e gestão das ações a realizar aquando a alteração do mesmo.
* ***FirebaseMessageHandler***: Este serviço gere o novo conteúdo vindo do serviço *GCM*. Aquando uma mensagem é recebida pelo serviço de *GCM*, gerido pelo sistema/*Google Play Services*, esse encarregar-se-á de comunicar a informação recebida com este serviço.
* ***ClipboardEventHandler***: Serviço comum ao serviço de *CopyMenuListener* e *FirebaseMessageHandler*. Tem como função a gestão do conteúdo recebido através dos outros serviços (copiar valores para o *clipboard* local, e/ou envio de valores para o *clipboard* partilhado). Este serviço não se encontra sempre em execução. Apenas é iniciado aquando um/ambos os serviços referidos anteriormente requerem o processamento de dados.

A comunicação entre os serviços pode ser descrita pela figura 4.



**Figura 4 – Comunicação entre serviços**

***Broadcast receivers***

A aplicação recorre a um *broadcast receiver* de forma a garantir que os serviços se iniciam aquando o dispositivo arranca.

***Content Provider***

Uma vez que a aplicação implementa um serviço de histórico seria necessário a utilização de um serviço para guardar os dados de forma persistente.

Para tal poderíamos recorrer a escrita dos dados num *content provider* ou recorrer a um simples ficheiro.

Optou-se pela utilização de um *content provider* devido a possibilidade de partilhar o seu conteúdo com outras aplicações e a garantia que caso exista múltiplas componentes a recorrer ao *content provider* existirá a garantia que ao ficará resolvido os possíveis problemas de sincronização.

**Deteção de alterações no *clipboard***

Para a deteção de alterações no *clipboard* do sistema *Android* recorreu-se à classe *ClipboardManager. O ClipboardManager trata-se de um serviço disponibilizado pelo sistema Android,* sendo obtido através do método *getSystemService,* no qual não é necessário a adição de nenhuma permissão de runtime e através deste é possível registar um *callback* controlado pela nossa aplicação que será executada quando o *clipboard* do sistema for alterado*.*

O callback ao ser invocado terá que processar a informação recebida. Caso seja concluído que a informação se trate de um formato suportado procederemos ao seu envio para o servidor e a sua classificação para ser guardado posteriormente no histórico.

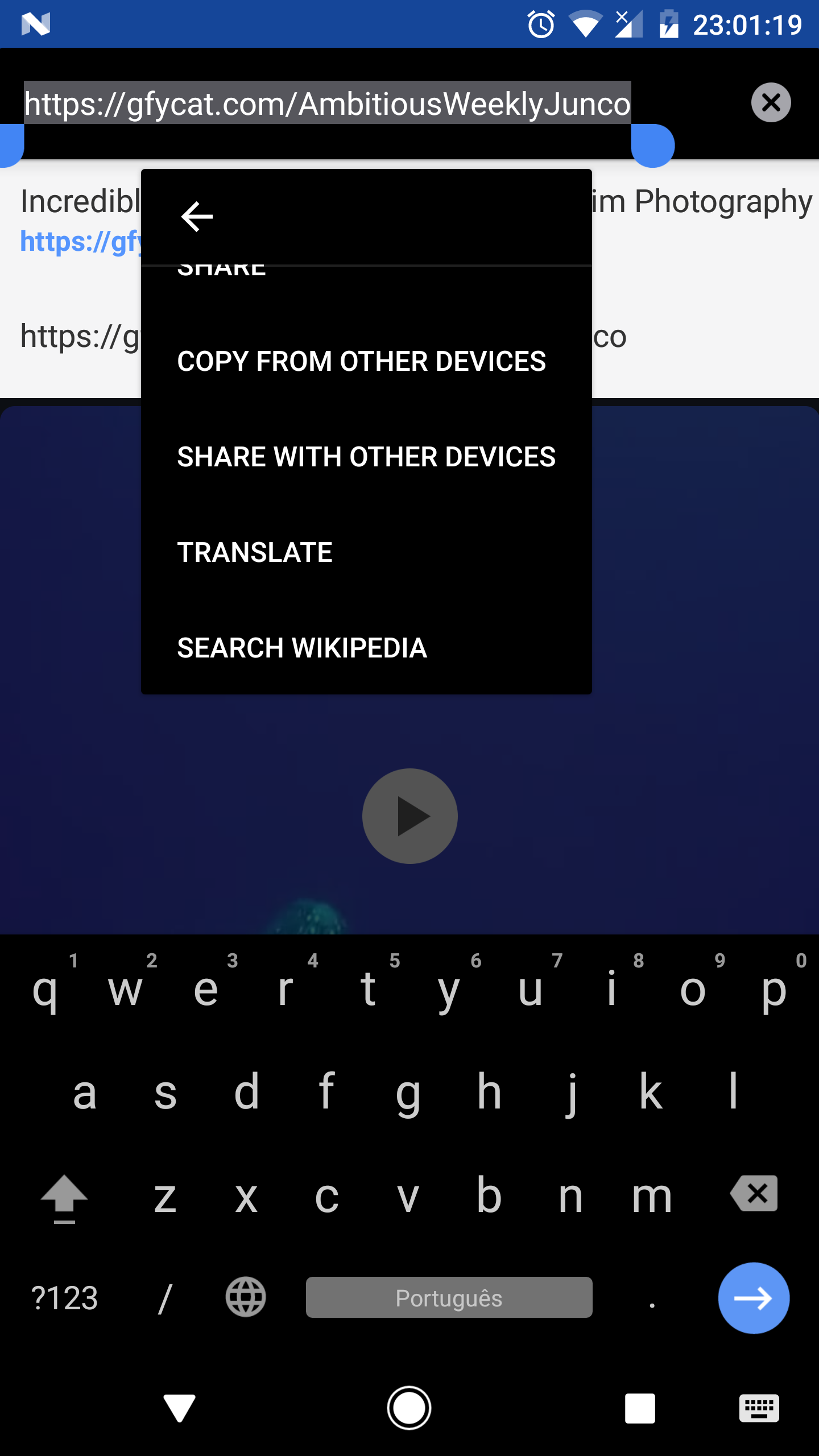
**Deteção de alterações no *clipboard* global (lado do servidor)**

Um dos pilares de boa programação em dispositivos móveis é minimizar tarefas de longo processamento.  
 Assim, desde o início do projeto foi descartada a ideia de estar constantemente a realizar uma operação de *pool* ao servidor para saber se o valor do *clipboard* global foi alterado.

Assim, as soluções pensadas para a resolução deste problema foram:

1. Recorrer a *API* disponibilizada pela versão 6.0 do *Android* (*API* 23) e acrescentar um botão de *paste,* no menu de copiar/colar (Figura 5 - Primeira abordagem para obtenção de dados do servidor), que quando pressionado pelo utilizador obteria o valor do conteúdo mais recente presente no servidor e guardava o mesmo no *clipboard* do dispositivo.
2. Outra solução passaria pela utilização dos serviços de acessibilidade do *Android*. Através deste seria possível definir uma ação ou várias ações no qual um *callback* seria invocado. Ponderou-se definir uma ação de *long press*, devido a este ser o método de invocação do menu de *copy/paste* em *Android*. Quando este callback fosse invocado proceder-se-ia a obtenção da informação que o servidor conteria. Apesar de esta solução ser melhor que 1), não seria a mais *user friendly* visto que os serviços de acessibilidade não foram feitos para estas situações e ter-se-ia que obrigar os utilizadores a ativar um serviço nas definições para a aplicação funcionar corretamente.
3. A última solução passaria por recorrer ao *Google Play Services*. Esta solução resolve os problemas do 1), visto que é suportado desde a *API* 10 do *Android*, é suportado nativamente e corretamente por todos os dispositivos e ainda é uma solução mais *user friendly* que 2).

Assim, o trabalho de obtenção de dados é feito por terceiros (*Google*), e caso exista novos dados a aplicação será notificada para a realização do seu processamento.



**Figura 5 - Primeira abordagem para obtenção de dados do servidor**

**Histórico**

Aquando a aplicação *Android* receber uma nova alteração do *clipboard*, tanto local como global, esta classifica o texto recebido como **texto, *links*** ou **contactos**. Após esta filtragem o texto é guardado num *Content Provider*, segundo a sua previa classificação, isto é o texto classificado como contato será guardado na tabela contatos, o texto classificado como *link* será guardo na tabela *links* e o texto sem classificação será guardado na tabela texto.

**Limitações**

Em seguida apresentam-se todas as limitações que foram aceites durante o desenvolvimento da componente *Android:*

* Impossibilidade de suporte para imagens devido a implementação do *clipboard* no *Android*. Devido a limitações presentes na *API* do *Android* não só a nossa aplicação teria que dar suporte para obtenção do ficheiro, como as aplicações de terceiros teria que ter suporte para obtenção da mesma. Visto que isto não acontece na maioria das aplicações a aplicação *Android* apenas suporta texto.
* Controlo da forma de obtenção de informação. Uma vez que a informação é obtida através do serviço *GCM* não é possível obter a informação **apenas** quando o utilizador tem *wifi* / dados móveis.
* Suporte para *API* >= 24. Uma vez que se pretendia utilizar novas funcionalidades oferecidas pela linguagem *Java* 8, optou-se por ter que limitar a aplicação para tal nível de *API*. Num ambiente de produção, onde é favorável englobar uma maior fatia de utilizadores seria necessário desenvolver para uma *API* inferior.

**Capítulo IV - Aplicação *Windows***

O desenvolvimento da aplicação *Windows* foi realizado recorrendo à linguagem *C#*.  
 Apesar de existirem vantagens em usar por exemplo a linguagem *Java*, desenvolver assim uma aplicação que fosse compatível com todos os sistemas operativos (partindo do princípio que estes tinham a *JVM* instalada), optou-se por usar *C#* e pela simplicidade de fazer pedidos *HTTP* assíncronos sem recorrer a *APIs* disponibilizadas por terceiros.

**Interação com o servidor**

Para a comunicação entre a aplicação *Windows* e o servidor, a aplicação não só recorre à *API fornecida por este como também ao serviço de WebSockets* disponibilizado pelo mesmo*.*

* ***API***

A aplicação recorre a *API* fornecida para realizar as ações de autenticação e para a obtenção de um *URI* de ligação de forma a poder estabelecer uma ligação com o servidor através de *WebSockets*.

* ***WebSockets***

Uma vez que não existe um serviço de *GCM* para dispositivos *desktop* o servidor implementa uma alternativa através de *WebSockets.*

A aplicação recorre aos *WebSockets* para obter nova informação, tal como a

submissão de novo conteúdo.

Para a comunicação através de *WebSockets* é necessário que a aplicação *Windows* realize as seguintes operações:

* Ao iniciar utiliza a *API* disponível para obter o *URI* do *socket* a conectar.
* Após obter o *URI* do *socket* será necessário que a aplicação envie uma mensagem *JSON* com a seguinte estrutura:

{

“action”: “register”,

“sub”: o id único do utilizador,

“id”: o id único do dispositivo

}

* Após este estar registado o servidor enviará todo o novo conteúdo automaticamente através do *socket*, e a aplicação poderá enviar novo conteúdo multimédia ou conteúdo textual através de uma mensagem *JSON* com os seguintes campos:

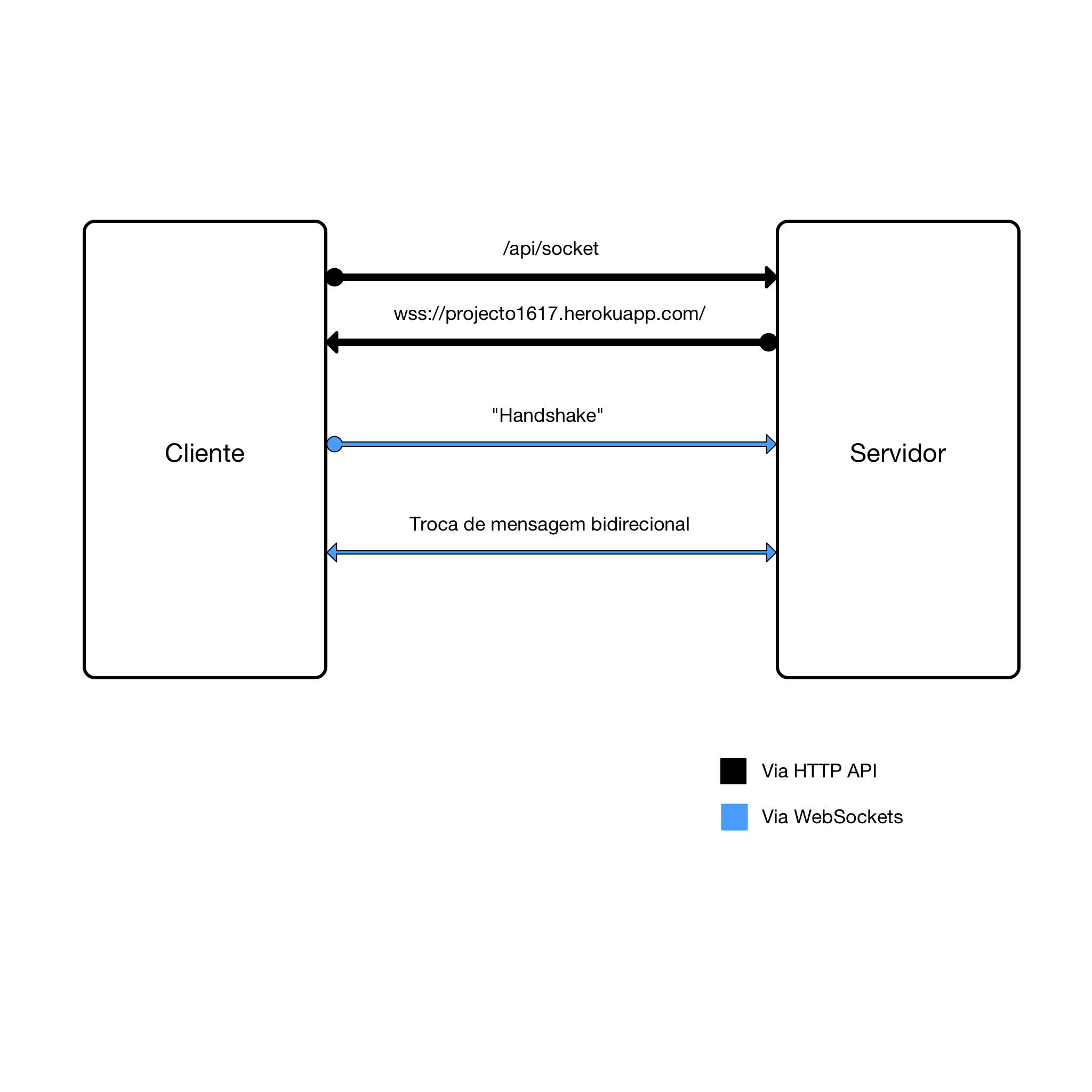
***action***: Cujo valor pode ser “*push*” ou “*pushMime*”, dependendo se o conteúdo se trata de uma informação textual ou um ficheiro, respetivamente.

**sub**: O id único do utilizador.

**device**: O id do dispositivo.

**data**: A informação em formato *string*. Caso se trate de um ficheiro deve converter o ficheiro para um *array* de *bytes* e de seguida codificar esse *array* de *bytes* numa *string* de *base64*.

***filename***: O nome do ficheiro, caso o pedido corresponda à de um pedido que contenha um ficheiro.

 A comunicação entre a aplicação e o servidor recorrendo à *websockets* pode ser descrita pela figura 6.

**Figura 6 – Comunicação utilizando WebSockets**

Como visto anteriormente para resolver os possíveis problemas de sincronização, o servidor por cada pedido *push* atribui um *id* único ao pedido e comunica-o ao cliente através de uma resposta ao pedido.

No caso da utilização da *API REST*, isto é facilmente gerido porque no protocolo *HTTP* existe sempre uma resposta para qualquer pedido.

No caso do protocolo de *Websockets*, essa noção não existe obrigatoriamente, contudo neste cenário é obrigatório existir.

Assim, aquando o fim do processamento do pedido feito o servidor comunicará com o cliente do pedido e dará uma resposta ao pedido com o seguinte formato *JSON*:

{

*“action”: “report”,*

*“title”: “success”,*

*“detail”: “success”,*

*“data”:* {

*“order”: n*

}

}

**Componentes**

No decorrer do desenvolvimento da aplicação *Windows* foram utilizados diversos componentes. Este subcapítulo serve para os identificar e explicar o motivo da sua utilização.

**Janelas**

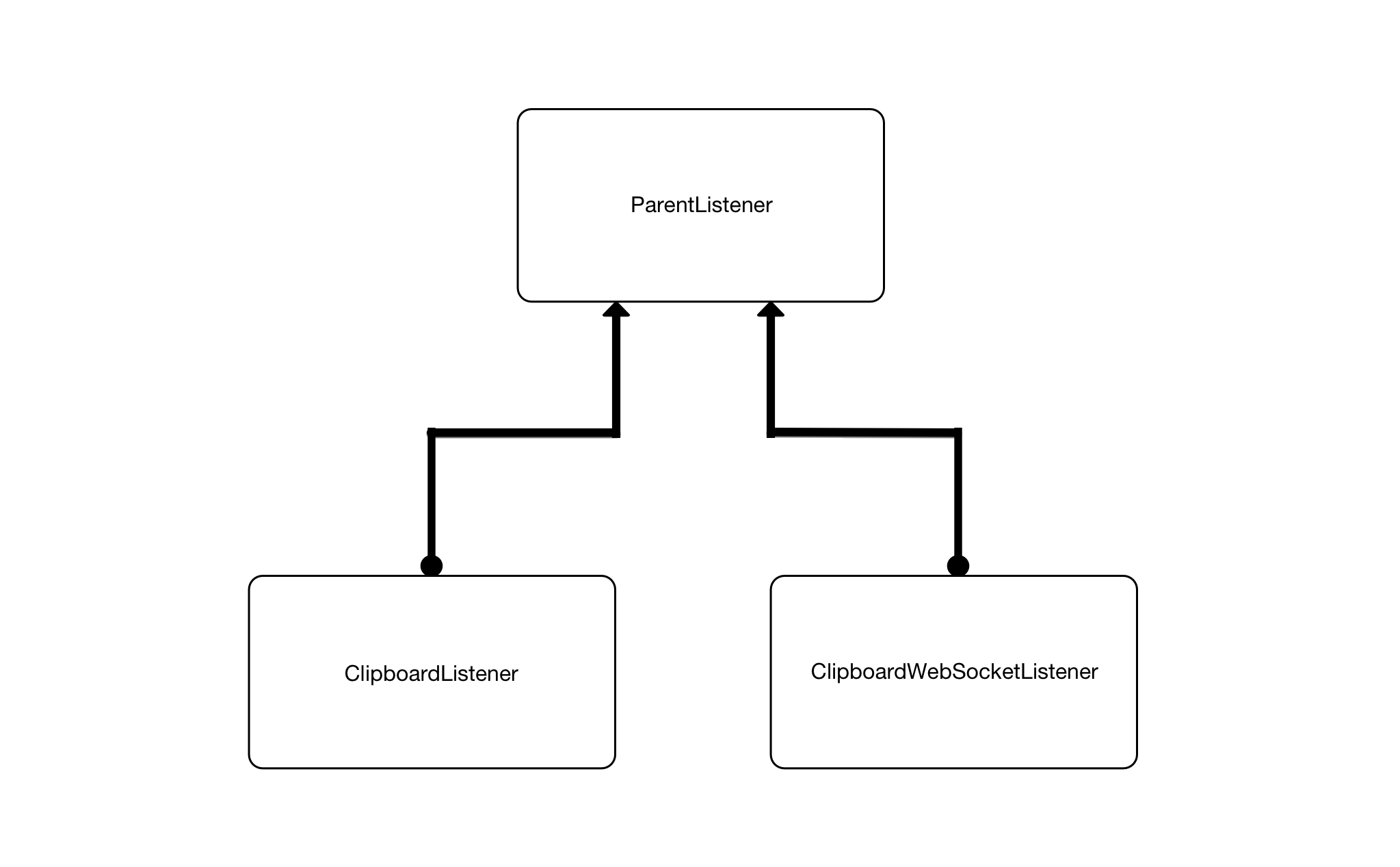
A aplicação *Windows* é composta por três janelas. Uma janela para realizar o *login*, outra para ajustar as definições do serviço/aceder ao histórico do dispositivo e ainda uma janela de “acerca” para o detalhe de licenças das bibliotecas utilizadas.

***Serviços***

Apesar de ser o mais indicado, no estado atual de implementação a aplicação não recorre a nenhum serviço. Em vez disso o trabalho está a ser realizado recorrendo a uma janela ou a *threads* que correm em *background*.

* ***ClipboardListener***: Janela utilizada para adicionar o *listener* ao *clipboard* do sistema. Foi utilizada uma janela, devido a necessidade de ao adicionar o *listener* é necessário especificar um *handler*.
* ***ClipboardWebSocketListener***: Thread que gere a comunicação de *websockets* entre o servidor e a aplicação.
* ***ParentListener***: “Serviço” em comum ao serviço de *ClipboardListener* e *ClipboardWebSocketListener*. Tem como função a gestão do conteúdo recebido através dos outros “serviços” (copiar valores para o *clipboard* local, e/ou envio de valores para o *clipboard* partilhado).

A comunicação entre estas três componentes pode ser descrita pela figura 7.



**Figura 7 – Comunicação entre serviços**

**Deteção de alterações no conteúdo do *clipboard***

Tal como o sistema *Android*, o *Windows* permite invocar um *callback* aquando a alteração do *clipboard* do sistema.

Para tal teremos que recorrer a duas funções pertencentes à *dll* *user32 (AddClipboardFormatListener* e *RemoveClipboardFormatListener*) e ainda fazer o *override* ao método *WndProc*, servindo este assim de forma a podermos ser notificados de qualquer alteração realizada ao *clipboard*.

Quando o método WndProc é invocado são realizadas as seguintes operações:

* Verificar se o código de ação corresponde ao código **0x031D,** caso contrário a ação realizada não foi a de copiar.
* Após verificarmos que o utilizador realizou a ação de copiar, obtemos o objeto ***IDataObject*** através da classe *clipboard* do *namespace* *System.Windows.Forms*.
* Verificamos se o objeto *IDataObject* é composto por formatos que a aplicação suporta, como por exemplo texto e imagens *jpg*.
* Após verificação, fazemos o *upload* da informação para o servidor recorrendo ao *WebSocket* previamente estabelecido.

**Deteção de alterações​ no *clipboard* global**

Ao contrário das aplicações móveis que tem preparado um serviço que realiza o trabalho de obtenção de informação automaticamente, esse serviço não se encontra disponível para as componentes *desktop*.

Numa primeira implementação optou-se por obter a informação recorrendo a *polling*, visto que os dispositivos *desktop* não precisam de ter os mesmos cuidados que as aplicações móveis, contudo, esta implementação tinha custos não só para o servidor uma vez que periodicamente teria que se estabelecer uma ligação com o servidor, como para o cliente uma vez que traria mais trabalho no desenvolvimento da aplicação *desktop*.

Assim, optou-se por desenvolver uma solução semelhante aos serviços disponibilizados pelo *GCM* recorrendo a *WebSockets*. Não só esta solução traz melhorias para a aplicação cliente como também para o servidor visto que este deixaria de ter que estabelecer ligações periodicamente.

**Histórico**

Uma vez que a aplicação *Windows* suporta texto e imagens esta recorre a maneiras distintas de gerir o histórico.

Caso a aplicação pretenda guardar texto no histórico, sendo que este veio do *clipboard* local ou do *clipboard* global, tal como a aplicação *Android* irá em primeiro lugar aplicar uma filtragem sobre o texto nas seguintes categorias **texto, *links*** ou **contatos.**

Uma vez catalogado e dado que não existem mecanismos como *Content Provider*, iremos guardar essa informação num ficheiro com o nome do filtro na pasta *roaming* do utilizador.

Caso a aplicação pretenda guardar uma imagem limitamo-nos a guardar a imagem, novamente, na pasta *roaming* do sistema, e mudamos o conteúdo do *clipboard* local para apontar para a nova localização do ficheiro.

**Limitações**

Em seguida apresentam-se todas as limitações que foram aceites durante o desenvolvimento da componente *Windows:*

* No decorrer do desenvolvimento da aplicação apenas se aceitou que os formatos multimédia sejam do tipo jpg.
* Utilização de serviços. Como referido anteriormente não foi utilizado nenhum serviço no decorrer deste projeto. A utilização de serviços traria como benefícios a possibilidade de ter o serviço em execução sem a necessidade de ter a aplicação em execução.

**Conclusão**

Os objetivos deste projeto foram compridos.

No decorrer deste projeto foi possível conhecer como é feita a gestão do *clipboard* de diferentes sistemas. No caso do Windows este trata de todo o conteúdo, incluído imagens, documentos, etc...

No caso do sistema *Android* isto pode variar. Caso se trate de uma versão de *Android* com a implementação da *Google* (*AOSP*), este apenas tratará de conteúdo do tipo texto e em caso de imagens este apenas guarda a localização para as mesmas, sendo que as aplicações que queiram recorrer ao conteúdo terão de tratar da obtenção da mesma. Contudo nem todas as versões de *Android* tem esta solução como sua implementação como o caso dos dispositivos *Samsung*. Este tem uma solução semelhante ao sistema *Windows*, onde as aplicações não terão que gerir a obtenção de recursos.

Foi também possível saber que tanto o *iOS* como o *macOs*, nativamente, não tem suporte para adicionar um *listener* ao *clipboard*.

Caso se pretenda por uma solução desta no mercado, não só ter-se-ia que rever a *API* da aplicação *Android*, como a utilização de serviços em *Windows* e ainda ponderar a utilização de um *load balancer* ao nível do servidor e possivelmente recorrer a um *cluster* de base de dados, de forma a distribuir a carga por diversas componentes.