UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tesi di laurea:

SVILUPPO DI UN SISTEMA PER L'INTEGRAZIONE DI APPLICAZIONI COME PLUGIN DI FIREFOX

Relatori:

Prof. Luigi Rizzo

Prof. Marco Avvenuti

Candidato:

Riccardo Panicucci

ANNO ACCADEMICO 2008/09

Alla mia famiglia

Indice

Introduzione **1**

1 Concetti generali sui plugin per Mozilla Firefox **3**
1.1 Che cosa è un plugin . . . 3
1.1.1 I vari tipi di plugin . . . 3
Plugin scriptabili . . . 4
1.1.2 Come visualizzare un plugin . . . 4
Il tag `object` . . . 5
Il tag `embed` . . . 6
1.1.3 Il ciclo di vita di un plugin . . . 6
1.2 NPAPI: le api per scrivere un plugin . . . 8
1.2.1 Inizializzazione e Distruzione . . . 9
`NP_Initialize()` . . . 9
`NP_Shutdown()` . . . 10
`NPP_New()` . . . 10
`NPP_Destroy()` . . . 11
`NPP_GetValue()` . . . 11
1.2.2 Gestione della finestra del plugin . . . 11
Le strutture dati . . . 11
`NPP_SetWindow()` . . . 12
1.2.3 Lo streaming con le `NPAPI` . . . 12
Strutture dati . . . 13
Le funzioni . . . 14
NPP_NewStream() . . . 15
NPP_WriteReady() . . . 15
NPP_Write() . . . 15
NPP_DestroyStream() . . . 15
NPP_StreamAsFile() . . . 15
1.2.4 Lo scripting . . . 15
Le strutture dati . . . 15

NPVariant 15
NPIdentifier 16
NPObject 16
NPClass 16
Le funzioni 16
NPN_CreateObject() 18
NPN_ReleaseObject() 18
NPN_RetainObject() 18
NPN_HasMethod()... 18
NPN_Evaluate() 18
NPN_ReleaseVariantValue() 18
NPN_GetStringIdentifier()... 18
NPN_IntFromIdentifier()... 18

2 Linee guida per lo sviluppo di un plugin 19
2.1 I files utilizzati 20
2.1.1 Files necessari per lo sviluppo di un plugin 20
2.2 Le macro introdotte 21
2.3 Linee guida 22

3 Descrizione del funzionamento di un plugin attraverso esempi 24
3.1 Le basi 24
3.1.1 Struttura dati usata 25
3.1.2 Inizializzazione e distruzione 25
3.1.3 Funzionamento 26
3.1.4 Compilazione 28
3.1.5 Uso di funzioni di inizializzazione personalizzate 29
3.2 I plugin scriptabili 29
3.2.1 L'esempio plugins1 29
Struttura dati usata 30
Funzionamento 30
3.2.2 L'esempio plugins2 31
Struttura dati usata 32
Inizializzazione 32
L'oggetto scriptabile 33
L'implementazione della NPClass 34
Il codice HTML per l'esempio 36

4 Le applicazioni SDL: esempi di integrazione in un plugin **37**
4.1 Breve introduzione a SDL . . . 37
4.1.1 Le finestre di SDL . . . 39
4.2 L'applicazione di esempio . . . 39
4.3 Lo sviluppo . . . 40
4.3.1 Il problema . . . 40
Il plugin . . . 40
Esecuzione dell'applicazione `sdlVanilla` . . . 40
4.3.2 Una prima soluzione . . . 41
Esecuzione . . . 42
4.3.3 La soluzione finale . . . 42
Esecuzione . . . 43

5 Il plugin app-wrapper **44**
5.1 Il file di configurazione . . . 44
5.2 Struttura dati . . . 46
5.3 A grandi linee . . . 46
5.4 Compilazione e Installazione . . . 47
5.5 Inizializzazione . . . 49
5.5.1 La funzione `NP_GetMIMEDescription()` . . . 49
5.5.2 Creazione dell'istanza . . . 50
5.5.3 Esecuzione del plugin . . . 50
5.5.4 Il ridimensionamento della finestra . . . 51

6 La versione per `Microsoft Windows` **53**
6.1 NPAPI: le differenze con Linux/FreeBSD . . . 53
6.1.1 Inizializzazione . . . 53
La funzione `NP_GetEntryPoints()` . . . 53
La funzione `NP_Initialize()` . . . 54
I MIME types . . . 54
6.2 Applicazioni SDL: embedding in una finestra esistente . . . 54
6.2.1 L'applicazione SDL . . . 55
6.3 Il plugin npbase.dll . . . 56
6.3.1 La funzione `NPP_SetWindow()` . . . 56
6.3.2 L'handler per l'evento `WM_SIZE` . . . 57

A Alcune funzioni in dettaglio **59**
A.1 La funzione `check_config()` . . . 59
A.1.1 Estrazione dei MIME types . . . 59
A.1.2 Estrazione delle opzioni e del comando da eseguire . . . 62

A.2 La funzione `plugin_calls_js()` 63

B Bibliografia **64**

Elenco delle figure

1.1 Icona plugin di default 6
1.2 Schema runtime plugin 7

4.1 Astrazione libreria SDL 38

5.1 Schema di appwrapper 48

Listings

1.1 Invocazione di un plugin attraverso il tag `object` 5
1.2 Invocazione di un plugin attraverso il tag `embed` 5
1.3 Esecuzione di un plugin a pagina intera. 5
1.4 Tag `object` nidificati 6
1.5 Attributi del tag embeb 8
1.6 `NP_Initialize()` 9
1.8 `NPP_New()` 10
1.7 `NP_Shutdown()` 10
1.9 `NPP_Destroy()` 11
1.10 `NPP_GetValue()` 11
1.11 Le strutture `NPWindow` e `NPSetWindowCallbackStruct` 12
1.12 `NPP_SetWindow()` 12
1.13 La struttura `NPStream` 13
1.14 Le funzioni riguardanti lo streaming 14
1.15 La struttura `NPObject` 16
1.16 La struttura `NPClass` 17
1.17 Alcune funzioni riguardanti lo scripting 17
3.1 La struttura dati _instance per il plugin1 24
3.2 Spezzone della funzione `NPP_New()` per il plugin1 25
3.3 La funzione `NP_GetMIMEDescription()` per il plugin1 25
3.4 Spezzone della funzione `NP_Destroy()` per il plugin1 25
3.5 Spezzone della funzione `NPP_GetValue()` per il plugin1 26
3.6 Spezzone della funzione `NPP_SetWindow()` per il plugin1 27
3.7 Output su terminale del plugin1 28
3.8 Modifiche al plugin1 per utilizzare funzioni di inizializzazione e shutdown personalizzate. 29
3.9 Spezzone della funzione `NPP_New()` per plugins1. 30
3.10 Spezzone della funzione `NPP_SetWindow()` per plugins1. 30
3.11 Frammenti di output per plugins1. 31
3.12 Struttura dati per plugins2. 32

3.13 Parte della funzione NPP_GetValue() per invocare la creazione dell'oggetto scriptabile, per plugins2. 33
3.14 La funzione my_GetScriptableObject() per la creazione dell'oggetto scriptabile, per plugins2. 33
3.15 L'implementazione della NPClass per l'oggetto scriptabile per plugins2. 35
3.16 Il codice HTML per l'esempio plugins2. 36
4.1 Codice HTML per eseguire l'applicazione sdlVanilla all'interno del plugin2. 40
4.2 Parte della funzione setwinevents() 41
5.1 File di configurazione *app-wrapper.conf* di esempio. 44
5.2 Struttura dati per il plugin app-wrapper. 45
5.3 La funzione NP_GetMIMEDescription() per il plugin app-wrapper. . 46
5.4 La funzione NPP_New() per il plugin app-wrapper. 49
5.5 La funzione NPP_SetWindow() per il plugin app-wrapper. 50
5.6 La funzione doResize() per il plugin app-wrapper. 51
6.1 Parte di codice dell'applicazione sdlwin.exe. 55
6.2 La funzione NPP_SetWindow() per il plugin npbase. 56
6.3 La funzione che agisce come handler per l'evento WM_SIZE 57
A.1 La ricerca dei MIME type supportati dal plugin appwrapper. 59
A.2 Estrazione delle opzioni e del comando da eseguire la file di configurazione appwrapper.conf. 62
A.3 Esecuzione di codice Javascript. 63

Introduzione

Questa tesi ha lo scopo di creare un plugin per il browser web Mozilla Firefox[1] che consenta di eseguire applicazioni arbitrarie all'interno di una pagina web.

Un plugin per Firefox è un componente software che espande le funzionalità del browser web, permettendo di eseguire un'applicazione esterna direttamente all'interno del browser. Per esempio, esistono plugin che consentono di visualizzare animazioni e filmati video di vario tipo nella pagina web, oppure plugin che permettono la visualizzazione di documenti pdf direttamente dal browser.

L'obiettivo iniziale era quello di inglobare un'interfaccia grafica per il centralino software `Asterisk`[2] all'interno di una pagina web. Questa interfaccia è stata realizzata usando la libreria grafica SDL. SDL è una libreria che crea uno strato astratto al di sopra di varie piattaforme software, e rende possibile la scrittura di applicazioni multimediali che possono girare su più sistemi operativi (Windows, Unix, ...) senza sostanziali modifiche.

Lo scopo finale di questa tesi è la creazione di un plugin, chiamato `app-wrapper`, in grado di inglobare nella pagina web vari tipi di applicazioni. Vedremo nel corso della tesi i differenti problemi che si incontrano nell'inglobare una generica applicazione SDL all'interno di un plugin, e mostreremo che sarà necessario apportare modifiche a queste applicazioni. Oltre alle applicazioni basate sulla libreria SDL, sarà possibile integrare nel plugin alcune applicazioni basate sulle librerie `X11`, quali il terminale *xterm* o il visualizzatore di video *mplayer*.

Lo sviluppo del plugin è stato ostacolato dalla presenza di documentazione relativamente obsoleta e fuorviante. Nonostante `Mozilla` metta a disposizione un'ampia documentazione sulle API per la scrittura dei plugin, gli esempi forniti utilizzano classi C++ per l'implementazione dell'istanza del plugin. Questo tipo di implementazione non è mai citata nella documentazione e lo sviluppatore si può trovare disorientato nel cercare di interpretare tali esempi.

Nella prima parte di questa tesi si intende presentare una documentazione chiara ed esauriente e un insieme di files sorgenti per la scrittura di un plugin. Per far ciò si

[1] http://www.mozilla.com/firefox/
[2] http://www.asterisk.org/

utilizzano le API originarie fornite da mozilla[3] senza sovrapporre alcuna classe C++. Inoltre, sono state realizzate alcune macro che aiutano lo sviluppatore nella scrittura del codice, evitando di scrivere funzioni di cui non necessita.

Lo sviluppo di un plugin è spiegato passo passo, e nel capitolo 3 vengono mostrati alcuni esempi via via più complessi. Successivamente sono introdotti i plugin *scriptabili*. Un plugin si dice *scriptabile* quando mette a disposizione variabili e funzioni accessibili dall'esterno del plugin stesso. Ad esempio, è possibile creare una pagina web che attraverso Javascript controlla l'esecuzione del plugin[4].

Successivamente si affronta il problema delle applicazioni SDL. Vengono utilizzati alcuni esempi che mostrano i problemi che ci sono nell'embedding di una applicazione SDL all'interno di un plugin[5], e le varie soluzioni affrontate. In particolare, non è possibile gestire gli eventi di tastiera e mouse, né ridimensionare la finestra dell'applicazione. Per risolvere questi problemi è necessario modificare l'applicazione SDL che si vuole eseguire all'interno del plugin. Nell'applicazione SDL è necessario recuperare alcune informazioni di basso livello riguardanti la finestra ed effettuare le chiamate necessarie per informare il `server X` che gli eventi di tastiera e mouse devono essere inviati all'applicazione SDL e non all'applicazione host[6].

Per consentire il ridimensionamento è necessario modificare l'applicazione SDL in modo tale che essa invii un evento di `ConfigureNotify` ogni volta che richiede un ridimensionamento. Il plugin deve essere in grado di catturare questo evento ed effettuare il resize della finestra. Per fare ciò, si utilizza la capacità di scripting del plugin, accedendo alle variabili `width` e `height` associate al plugin stesso.

Il capitolo 5 affronta lo sviluppo del plugin `app-wrapper`. Questo plugin sfrutta le soluzioni descritte nei capitoli precedenti, permettendo l'esecuzione di applicazioni basate sulla libreria SDL e di alcune applicazioni di X che hanno la possibilità di essere eseguite in una finestra già esistente. Ad esempio, è possibile eseguire l'applicazione `xterm` o il player multimediale `mplayer`, permettendo di gestire tutti gli eventi di mouse e tastiera e di ridimensionare la finestra.

Infine nell'ultimo capitolo sono evidenziate le differenze principali per sviluppare un plugin per `Microsot Windows`. Viene presentata la soluzione adottata per permettere l'esecuzione di applicazioni SDL all'interno di una finestra esistente, e infine si mostra un prototipo del plugin `app-wrapper` che gira in ambiente `Microsoft Windows`, il quale permette attualmente soltanto l'esecuzione di applicazioni SDL.

[3] https://developer.mozilla.org/en/Gecko_Plugin_API_Reference

[4] pensiamo per esempio ad un player multimediale. É possibile creare pulsanti in una pagina web per controllare la riproduzione del video...

[5] il medesimo tipo di problema si ha nell'inglobare la stessa applicazione in una generica finestra già esistente.

[6] che ha creato la finestra, nel nostro caso il plugin.

Capitolo 1

Concetti generali sui plugin per Mozilla Firefox

1.1 Che cosa è un plugin

Un plugin per Mozilla firefox è una libreria di codice nativo (C, C++) che gira solo all'interno del browser. I plugin offrono un'ampia varietà di caratteristiche che consentono di ampliare la flessibilità del browser. Esistono attualmente molti plugin, alcuni dei quali molto diffusi, che consentono di vedere all'interno della pagina web documenti pdf[1], visualizzare video[2] o animazioni Flash[3], e molti altri ancora. Per controllare i plugin installati nel sistema basta digitare `about:plugins` nella barra degli indirizzi del browser: comparirà una pagina che elenca tutti i plugin installati, con il proprio MIME type gestito (uno o più) e l'estensione associata.

1.1.1 I vari tipi di plugin

Ci sono due principali tipi di plugin:

Un **windowed plugin** è un plugin che è disegnato nella sua finestra nativa. Il browser fornisce un `id` (che dipende dalla piattaforma usata) che identifica la finestra nella quale il plugin gira. Questa finestra può essere una porzione[4] della pagina web oppure una pagina intera[5]. Gli eventi e il disegno di questa finestra sono

[1] Adobe Acrobat Reader
[2] Mplayer, Windows Media Player...
[3] Adobe Flash
[4] Per esempio un visualizzatore di video
[5] per esempio la visualizzazione di un documento pdf

gestiti dal plugin utilizzando le tecniche fornite dal sistema operativo. In particolare, su Windows il browser registra una *window class* e crea un'istanza di quella classe per il plugin. Il plugin può quindi effettuare un *subclassing* della finestra per essere in grado di gestirne tutti gli eventi.

Su Unix il browser crea un widget di tipo *Motif* e passa il *Window ID* al plugin. É passata anche una struttura che consente di accedere a basso livello a questo widget in modo tale da poterne gestire gli eventi.

Un **windowless plugin** è un plugin che non richiede una finestra nativa. È disegnato in un target chiamato *drawable* che può essere sia la finestra del browser sia una bitmap posizionabile all'interno della pagina web. Il browser in questo caso è responsabile di controllare sia il ridisegno che la gestione degli eventi.

La maggioranza dei plugin attualmente in circolazione sono plugin windowed. Il supporto ufficiale ai plugin windowless è stato aggiunto nelle versioni Unix di Firefox soltanto nelle ultime release[6].

Plugin scriptabili

Un plugin si dice *scriptabile* se è in grado di accedere agli oggetti del browser[7], e di esportare funzioni e variabili del plugin all'esterno e renderli disponibili al browser[8]. Un plugin di tale tipo mette a disposizione del browser un oggetto il quale contiene variabili e funzioni che possono essere usati dall'esterno. Per esempio un plugin che agisce come player video può mettere a disposizione di Javascript alcune funzioni in grado di mettere in pausa il video o scorrere attraverso esso. Sarà quindi possibile creare nella pagina web pulsanti in grado di controllare la visione del video.

1.1.2 Come visualizzare un plugin

A seconda di come il plugin è invocato nel codice html della pagina web, esso può essere visualizzato come una pagina intera[9] oppure essere integrato nella pagine web[10]. Per invocare un plugin si usano i tag html `object` o `embed`. Per esempio i codici 1.1 e 1.2 possono essere utilizzati per integrare un filmato in una pagina web.

[6] a partire da Firefox 3.0

[7] in generale è possibile accedere a tutto il DOM *(Document Object Model)*, come se si fosse all'interno di una funzione *Javascipt*

[8] e poterli invocare all'interno di una funzioni Javascript

[9] come il caso del visualizzatore pdf di Adode

[10] ad esempio per un animazione flash o un video

Listing 1.1: *Invocazione di un plugin attraverso il tag* `object`

```
<object data="test.avi" type="video/avi"
        width="320" height="200" id="test">
</object>
```

Listing 1.2: *Invocazione di un plugin attraverso il tag* `embed`

```
<embed src="test.avi" type="video/avi"
       width="320" height="200" id="test">
```

É interessante notare il nome dell'attributo del file da visualizzare, che cambia da `data` a `src` a seconda del tag usato. Sono stati inseriti anche attributi per dimensionare la finestra del plugin. Gli attributi non riconosciuti dal browser sono passati all'istanza del plugin nel momento in cui viene creata. Tramite il valore dell'attributo `id` è possibile riferirsi all'istanza del plugin e in caso di plugin scriptabile tramite questo identificatore è possibile accedere ai membri ed effettuare chiamate a funzioni del plugin. Invocando il tag `object` senza gli attributi `width` e `height` (codice 1.3) il plugin viene eseguito, se possibile,[11] a pagina intera.

Listing 1.3: *Esecuzione di un plugin a pagina intera.*

```
<object data="test.pdf">
<\object>
```

Il tag `object`

In generale è utilizzato per inglobare una varietà di tipi di oggetti (immagini, applet java, plugin) in una pagina web. Inizialmente era previsto l'utilizzo del tag `embed` per inglobare i plugin in una pagina web, ma l'utilizzo del tag `object` fornisce molta più flessibilità.

Il codice 1.4 mostra un esempio di utilizzo di tag `object` nidificati. Per prima cosa viene elaborato il tag `object` più esterno: se il plugin Silverlight[12] è installato viene eseguito, altrimenti il browser elabora il tag `object` più interno e tenta di eseguire il plugin Flash. Da notare l'attributo `codebase` che fornisce un link dal quale poter scaricare il plugin Flash se non è presente nel sistema.

[11] Un plugin deve essere sviluppato in modo tale da supportare l'esecuzione a pagina intera, specificandolo nel momento in cui viene chiamata la funzione `NPP_New()`.

[12] Un implementazione di Microsoft simile ad Adobe Flash

Listing 1.4: *Tag* `object` *nidificati*

```
[...]
<object data="foo.scr"
        type="application/x-silverlight"
        width="300"
        height="240">
    <object data="foo.swf"
        type="application/x-shockwave-flash"
        width="300"
        height="240"
        codebase="http://www.macromedia.com/[...]"/>
    </object>
</object>
[...]
```

Figura 1.1: *Icona plugin di default, visualizzata quando il plugin richiesto non è installato nel sistema e si è utlizzato il tag* `embed` *nel codice html.*

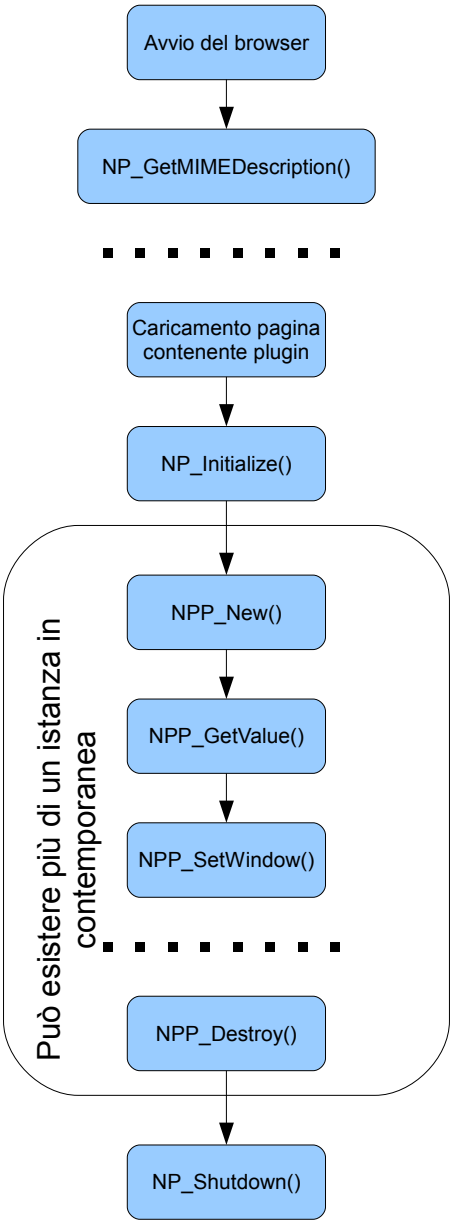
Il tag `embed`

Nonostante che il tag `object` sia il metodo preferito per integrare plugin in una pagina web, spesso è utilizzato il tag `embed` per questioni di compatibilità con browser vecchi e per chiedere esplicitamente all'utente l'installazione di un plugin mancante, in quanto il plugin di default (vedi figura 1.1) è invocato automaticamente soltanto quando si utilizza il tag `embed`. É necessario specificare almeno l'atttributo `src` o `type` (vedi il codice 1.5) in modo tale che il browser possa capire quale plugin è necessario invocare. Gli attributi non riconosciuti dal browser saranno passati al plugin.

1.1.3 Il ciclo di vita di un plugin

Il ciclo di vita di un plugin è controllato dalla pagina web che lo invoca. Nel momento in cui il browser viene avviato, è controllata la presenza di nuovi plugin in alcuni percorsi standard. In Linux/FreeBSD sono controllate sia le directory che contengono i plugin comuni a tutti gli utenti (ad esempio in Gentoo Linux `/usr/lib/nsbrowser/plugins/`) sia la directory `~/.mozilla/plugins` che contiene i plugin relativi ai singoli utenti. Quando il browser trova nuo-

Figura 1.2: *Schema del ciclo di vita di un plugin generico.*



Listing 1.5: *Attributi del tag embeb*

```
<embed
  src="location"
  type="mimetype"
  pluginspage="instrUrl"
  pluginurl="pluginUrl"
  align="left"|"right"|"top"|"bottom"
  border="borderWidth"
  frameborder="no"
  height="height"
  width="width"
  units="units"
  hidden="true|false"
  hspace="horizMargin"
  vspace="vertMargin"
  name="pluginName"
  palette="foreground"|"background"
 >
</embed>
```

vi plugin (o cambiamenti del timestamp dall'ultimo avvio), invoca la funzione `NP_GetMIMEDescription()` che restituisce il tipo di oggetto (MIME type) gestito dal plugin. Quando un utente apre una pagina web che contiene oggetti che invocano un plugin, il browser esegue le seguenti azioni (vedi figura 1.2):

- controlla se esiste un plugin in grado di gestire il MIME type indicato;
- carica il plugin in memoria;
- inizializza il plugin;
- crea una nuova istanza del plugin.

Il browser può caricare in contemporanea più istanze dello stesso plugin nella stessa pagina oppure in pagine diverse. Quando l'utente lascia la pagina (o chiude la finestra) l'istanza del plugin è distrutta. Quando l'ultima istanza del plugin viene distrutta, il browser scarica dalla memoria il codice del plugin. A questo punto un plugin non occupa nessuna risorsa di sistema eccetto lo spazio su disco.

1.2 NPAPI: le api per scrivere un plugin

Mozilla mette a disposizione la *Netscape Plug-in Application Programming Interface*[13] (in breve NPAPI), composta da due gruppi di funzioni e un insieme di strutture dati

[13] https://developer.mozilla.org/en/Gecko_Plugin_API_Reference

- Funzioni che iniziano per `NPP_*`: sono le funzioni che il plugin implementa e il browser usa;
- Funzioni che iniziano per `NPN_*`: sono le funzioni implementate dal browser e che il programmatore può utilizzare.
- Strutture dati: tipi specifici per la scrittura di un plugin. Iniziano per `NP*`, sono definiti nel file `npapi.h` fornito da Mozilla.

Esistono alcune funzioni[14] che sono gli *entry point* della libreria, non fanno parte di nessuna di queste categorie e non sono correlati con nessuna istanza del plugin.

Un plugin è una libreria di codice nativo per una specifica piattaforma. NPAPI è stata sviluppata in modo tale da essere multipiattaforma. In generale un plugin sviluppato per un sistema operativo può essere portato senza troppe complicazioni su un altro sistema, facendo attenzione a non usare costrutti specifici di un sistema.

In unix un plugin deve avere estensione `*.so` o `*.dso`, mentre in Windows il nome del file deve avere un formato del tipo `NP*.dll`.

Vediamo adesso con maggior dettaglio alcune delle funzioni facenti parti delle `NPAPI`.

1.2.1 Inizializzazione e Distruzione

In questo gruppo sono presenti le funzioni che si occupano dell'inizializzazione e della creazione delle istanze del plugin, e in contrappossizione di effettuare tutte le operazione di distruzione del plugin stesso. In una pagina web è possibile avere più istanze di uno stesso plugin. Esse possono condividere una parte dello spazio di lavoro.

`NP_Initialize()`

É la funzione che esegue le inizializzazioni globali per un plugin. É chiamata dal browser prima della creazione della prima istanza del plugin.

Listing 1.6: *`NP_Initialize()`*

```
NPError
NP_Initialize(NPNetscapeFuncs *aNPNFuncs,
              NPPluginFuncs *aNPPFuncs)
```

Il parametro `NPNetscapeFuncs*` contiene una lista di puntatori alle funzioni `NPN_*` sono fornite dal browser e che andrà salvato in una variabile globale, mentre il parametro `NPPluginFuncs*` contiene i puntatori alle funzioni `NPP_*`, implementate dal

[14] `NP_Initialize()`, `NP_Shutdown()`, `NP_GetMIMEDescription()`

Listing 1.8: *NPP_New()*

```
NPError
NPP_New(NPMIMEType pluginType, /* MIME type della nuova istanza */
        NPP instance, /* Puntatore alla nuova istanza. E' creato dal browser,
                         e' possibile memorizzare qui dati privati
                         appartenenti all'istanza del plugin. */
        uint16 mode, /* NP_EMBED: istanza creata attraverso il tag EMBED, e'
                                  integrato nella pagina web;
                        NP_FULL:  l'istanza e' visualizzata a schermo intero.*/
        int16 argc, /* Numero di argomenti passati al plugin. */
        char *argn[], /* Array dei nome dei parametri passati. */
        char *argv[], /* Array del valori del parametri passati. */
        NPSavedData *saved /* Puntatore a dati salvati da un'istanza precedente
        . */);
```

plugin, e che saranno esportate in modo tale che il browser le possa chiamare quando necessario. In questa funzione è possibile allocare spazio per eventuali variabili globali condivise fra tutte le istanze del plugin.

`NP_Shutdown()`

Listing 1.7: *NP_Shutdown()*

```
void
NP_Shutdown(void)
```

È la funzione che il browser chiama quando viene distrutta l'ultima istanza del plugin. Dopo questa chiamata il codice del plugin non è più presente in memoria. È necessario liberare eventuale memoria allocata in `NP_Initialize()`.

`NPP_New()`

É chiamata dal browser per la creazione di una nuova istanza. Il puntatore alla struttura `NPP` è valido finchè non viene chiamata la funzione `NPP_Destroy()` ed è possibile memorizzare qui le informazioni private relative all'istanza del plugin. Viene inoltre passato il MIME type, il modo di visualizzazione (full-screen o embedded), e le informazioni sui parametri passati al plugin tramite il tag `embed` o `object`. É possibile recuperare eventuali dati salvati dalla funzione `NPP_Destroy()` di un istanza precedente attraverso il puntatore `saved`.

Listing 1.9: *NPP_Destroy()*

```
NPError
NPP_Destroy(NPP instance,
            NPSavedData **save );
```

Listing 1.10: *NPP_GetValue()*

```
NPError
NPP_GetValue(NPP instance, NPPVariable variable, void *value);
```

`NPP_Destroy()`

Elimina un'istanza del plugin. Vengono eliminati tutti i dati e le risorse allocate all'istanza del plugin. Il browser chiama questa funzione quando lascia la pagina contenente il plugin, chiude la pagina o il browser. Se non esistono altre istanze del plugin il browser chiama la funzione `NP_Shutdown()`. É possibile salvare dati nel parametro `save` da poter riutilizzare se l'utente ritorna alla pagina appena lasciata. Non è consigliabile salvare dati critici qui in quanto il browser potrebbe eliminare questo buffer ogniqualvolta lo ritenga oppurtuno.

`NPP_GetValue()`

Il browser invoca questa funzione quando vuole chiedere informazioni riguardanti il plugin. In particolare, nel parametro `variable` è presente un identificatore per il tipo di informazione richiesta, che sarà passata nel puntatore `value`. Per esempio, attraverso la variabile `NPPVpluginScriptableNPObject` si informa il browser dell'utilizzo di un oggetto scriptabile, e lo si crea ritornandolo nella variabile `value`. Per un elenco di tutte le caratteristiche fornite si vedano gli esempi forniti più avanti nel testo.

1.2.2 Gestione della finestra del plugin

Le strutture dati

La struttura dati principale è `NPWindow` (codice 1.11) che contiene le informazioni riguardanti la geometria della finestra e l'identificatore assegnato (che dipende dalla piattaforma in uso). Inoltre se si sta usando un sistema operativo `Linux/FreeBSD` la struttura `NPWindow` contiene un puntatore a una struttura `NPSetWindowCallbackStruct` contenente ulteriori parametri della finestra, quali il *Display* e altri attributi della finestra.

Listing 1.11: *Le strutture NPWindow e NPSetWindowCallbackStruct*

```
typedef struct {
    /* Contiene vari parametri relativi alla finestra dell'ambiente X-Window */
    int32          type;
    Display*       display;
    Visual*        visual;
    Colormap       colormap;
    unsigned int   depth;
} NPSetWindowCallbackStruct;

typedef struct _NPWindow {
    void*     window; /* Identificatore della finestra. */
    uint32    x;      /* Coordinate dell'angolo superiore a sinistra  */
    uint32    y;      /* relative alla finestra del browser. */
    uint32    width;  /* Larghezza della finestra. */
    uint32    height; /* Altezza delle finestra. */
    [...]
#ifdef XP_UNIX
    void *    ws_info; /* Puntatore a NPSetWindowCallbackStruct. */
#endif
} NPWindow;
```

La struttura `NPWindow` è passata alla funzione `NPP_SetWindow()` che si occupa del disegno della finestra del plugin.

`NPP_SetWindow()`

Listing 1.12: *NPP_SetWindow()*

```
NPError
NPP_SetWindows(NPP instance,
               NPWindow *window );
```

Il browser chiama questa funzione dopo aver creato l'istanza per consentire di iniziare il disegno della propria finestra. Può venire chiamata successivamente se la finestra cambia posizione o dimensione, oppure quando è necessario effettuare un ridisegno della stessa[15]. L'oggetto `NPWindow` (vedi codice 1.11) contiene le coordinate della finestra e altri parametri relativi alla piattaforma in uso. Questa finestra sarà valida finchè non viene chiamata questa funzione con un oggetto NPWindow diverso.

1.2.3 Lo streaming con le `NPAPI`

Uno stream è un oggetto che rappresenta un URL e i dati ad esso associati. Può essere prodotto dal browser e consumato dal plugin e viceversa. Ogni stream ha associato

[15] Ad esempio quando una parte della finestra torna visibile dopo essere stata nascosta da un'altra

un MIME type[16] che identifica il formato dei dati dello stream. Uno stream prodotto dal browser può essere sia inviato automaticamente al plugin che essere richiesto esplicitamente dal plugin a seconda del metodo di trasmissione selezionato.

Il browser informa il plugin attraverso la funzione `NPP_NewStream()` che è presente un nuovo stream. É possibile selezionare[17] il metodo di trasferimento per gli stream prodotti dal browser fra tre modalità distinte a seconda delle necessità:

1. Ricevere lo stream automaticamente nel momento in cui i dati arrivano. Dopo la chiamata a `NPP_NewStream()` il browser effettua le chiamate alle funzioni `NPP_WriteReady()` per vedere quanti bytes il plugin può accettare e a `NPP_Write()` per inviarle. Quanto l'invio è terminato (oppure sono avvenuti errori[18]) il browser chiama la funzione `NPP_DestroyStream()`.

2. Accedere ai dati dello stream in modo casuale. Per fare questo il browser salva l'intero[19] stream in un file temporaneo e lo rende accessibile al plugin attraverso la funzione `NPN_RequestRead()`.

3. Salvare il file in un percorso locale. Il browser chiama la funzione `NPP_StreamAsFile()` per passare al plugin il percoso completo del file.

Strutture dati

Listing 1.13: *La struttura `NPStream`*

```
typedef struct NPStream
{
    void* pdata;
    void* ndata;
    const char* url;
    uint32      end;
    uint32      lastmodified;
    void*       notifyData;
    const char *headers;
} NPStream;
```

La struttura `NPStream` (codice 1.13) rappresenta uno stream. Il browser alloca e inizializza questa struttura e la passa come parametro alle funzioni `NPP_NewStream()` o `NPN_NewStream()` a seconda che lo stream è creato dal browser o dal plugin.

[16] un plugin può gestire più MIME types. Per esempio un plugin che gestisce il video può trattare video in vari formati.

[17] vedi `NPP_NewStream()`

[18] errori di rete, o annullamento dell'operazione da parte dell'utente

[19] Ciò può essere evitato se il server HTTP supporta l'estensione "byte-range" per richiedere un intervallo preciso di dati

Listing 1.14: *Le funzioni riguardanti lo streaming*

```
NPError NPP_NewStream(NPP instance, NPMIMEType type, NPStream* stream,
                      NPBool seekable, uint16* stype);

int32 NPP_WriteReady (NPP instance, NPStream *stream);

int32 NPP_Write (NPP instance, NPStream *stream, int32 offset, int32 len,
                 void *buffer);

NPError NPP_DestroyStream (NPP instance, NPStream *stream, NPError reason);

void NPP_StreamAsFile (NPP instance, NPStream* stream, const char* fname);
```

Uno stream è valido finchè non sono chiamate le funzioni `NPP_DestroyStream()` o `NPN_DestroyStream()`.

pdata puntatore che può essere utilizzato per memorizzare informazioni addizionali a questo stream. Il browser non modifica questo campo;

ndata puntatore per memorizzare informazioni private sullo stream. Utilizzato dal browser, il plugin non deve modificarlo;

url contiene l'indirizzo completo dello stream

end lunghezza dello stream. Può essere 0 se non è conosciuta;

lastmodified data dell'ultima modifica dell'url, in secondi a partire dal 1 gennaio 1970;

notifyData utilizzato solo dagli stream generati attraverso le funzioni `NPN_GetURLNotify()` e `NPN_PostURLNotify`, contiene il parametro passato a tali funzioni;

headers usato solo per stream HTTP e contiene l'header HTTP passato dal server.

Le funzioni

Nel codice 1.14 sono presenti alcune funzioni riguardanti lo streaming. Di seguito viene fornita una breve descrizione, per eventuali approfondimenti si veda la documentazione fornita nei files sorgenti `myplugin.c` e `npn.c`.

NPP_NewStream() Il browser informa il plugin della presenza di un nuovo stream. Lo stream può essere specificato nell'attributo `src` o `data` dei tag `embed` o `object` rispettivamente. Il parametro `stype` definisce il modo di trasferimento dello stream, che può essere uno dei tre descritti in precedenza.

NPP_WriteReady() Il browser chiama questa funzione ogni volta prima di chiamare `NPP_Write()` per conoscere il numero di bytes che l'istanza può ricevere.

NPP_Write() Il browser spedisce al massimo `len` bytes all'istanza del plugin. Il numero di bytes effettivamente spediti è ritornato dalla funzione. Un numero negativo indica che c'è stato un qualche tipo di errore, in conseguenza di ciò il browser chiama `NPP_DestroyStream()`.

NPP_DestroyStream() Il browser informa il plugin che lo stream sta per essere eliminato. Il parametro `reason` indica se ci sono stati errori o no.

NPP_StreamAsFile() Il browser chiama questa funzione per informare il plugin del percorso locale dove è stato salvato lo stream.

1.2.4 Lo scripting

Come detto in precedenza un plugin *scriptabile* mette a disposizione del browser (e di conseguenza a *Javascript*) funzioni e variabili.

Per fare questo, il plugin attraverso la funzione `NPP_GetValue(npp, NPPVpluginScriptableNPObject, ...)` informa il browser che vuole esporre un oggetto.

Le strutture dati

Per supportare lo scripting le `NPAPI` sono state espanse, e nuove strutture dati e funzioni sono state introdotte.

NPVariant É una struttura che mantiene un valore e il tipo di quel valore. Per manipolare questa struttura è necessario usare le funzioni e alcune macro messe a disposizione dalle NPAPI:

- macro per controllare se un valore è di un certo tipo (`NPVARIANT_IS_BOOLEAN()`, `NPVARIANT_IS_INT32()`, `...`)

- macro per estrarre un valore da un NPVariant (NPVARIANT_TO_INT32(), NPVARIANT_TO_STRING(), ...)
- macro per inizializzare un NPVariant (INT32_TO_NPVARIANT(), BOOLEAN_TO_NPVARIANT(), ...)

NPIdentifier É un tipo opaco usato per gli identificatori di metodi e funzioni. Gli NPIdentifier sono unici e il tempo di vita è controllato dal browser.

NPObject Un NPObject rappresenta l'oggetto scriptabile che viene esposto a Javascript.

Listing 1.15: *La struttura NPObject*

```
struct NPObject {
  NPClass *_class;
  uint32_t referenceCount;
  /* Additional space may be allocated here by types of NPObjects */
};
```

La struttura mantiene un puntatore a una NPClass e un contatore di oggetti referenziati. É possibile memorizzare in questa struttura ulteriori dati privati riguardanti l'oggetto, come ad esempio un puntatore all'istanza del plugin, come utilizzato negli esempi dei prossimi capitoli.

Poichè gli NPObject sono oggetti referenziati è necessario chiamare le appropiate funzioni delle API ogni volta che si intende utilizzare un oggetto.

Il browser espone sempre il proprio oggetto *window* cosicchè è possibile accedere a tutto il DOM dall'interno del plugin.

NPClass La struttura NPClass contiene un insieme di puntatori alle funzioni che implementano l'oggetto scriptabile. É necessario implementare tutte queste funzioni in modo tale che il browser non trovi nessun puntatore nullo in questa struttura. Per eseguire le operazioni sull'oggetto scriptabile non si devono utilizzare queste funzioni, ma effettuare chiamate attraverso le funzioni messe a disposizione dalle NPAPI.

Le funzioni

Vediamo adesso alcune delle funzioni principali riguardanti lo scripting. Per un elenco completo delle funzioni, si vedano i file sorgenti forniti[20] e la documentazione di Mozilla.[21]

[20] myplugin.h e npn.c

[21] https://developer.mozilla.org/en/Gecko_Plugin_API_Reference/Scripting_plugins

Listing 1.16: *La struttura `NPClass`*

```
static NPClass xyz_scriptable =
{
        .structVersion = NP_CLASS_STRUCT_VERSION,
        .allocate = _allocate,
        .deallocate = _deallocate,
        .invalidate = _invalidate,
        .hasMethod = _hasMethod,
        .invoke = _invoke,
        .hasProperty = _hasProperty,
        .getProperty = _getProperty,
        .setProperty = _setProperty,
        .removeProperty = _removeProperty,
        .invokeDefault = _invokeDefault,
};
```

Listing 1.17: *Alcune funzioni riguardanti lo scripting*

```
NPObject *NPN_CreateObject(NPP instance, NPClass *aClass);
void NPN_ReleaseObject(NPObject *obj);
NPObject *NPN_RetainObject(NPObject *obj);
bool NPN_HasMethod(NPP npp, NPObject* obj, NPIdentifier methodName);
[...]
bool NPN_Evaluate(NPP npp, NPObject* obj, NPString *script, NPVariant *result);
void NPN_ReleaseVariantValue(NPVariant *variant);
NPIdentifier NPN_GetStringIdentifier(const NPUTF8 *name);
[...]
int32_t NPN_IntFromIdentifier(NPIdentifier identifier);
[...]
```

`NPN_CreateObject()` Crea un nuovo oggetto scriptabile. Se la `NPClass` passata come parametro contiene una funzione `_allocate()` viene invocata, altrimenti la memoria è allocata tramite una `malloc()`. Il contantore del nuovo oggetto `NPObject` è inizializzato a 1 prima di ritornare.

`NPN_ReleaseObject()` Decrementa il contatore dell'oggetto referenziato, e se raggiunge 0 l'oggetto viene deallocato chiamando la funzione `_deallocate()` se fornita, oppure semplicemente con una `free()`.

`NPN_RetainObject()` Informa il browser che l'oggetto specificato è utilizzato, e ne incrementa il contatore.

`NPN_HasMethod()...` Questa funzione, insieme ad altre,[22] è utilizzata per eseguire le funzioni implementate nella `NPClass`.

`NPN_Evaluate()` Esegue uno script nello *scope* dell'oggetto `obj`. L'eventuale risultato è passato nel parametro `result`. Il chiamante deve invocare la funzione `NPN_ReleaseVariantValue()` quando non ha più bisogno del risultato.

`NPN_ReleaseVariantValue()` Dealloca il valore della `NPVariant` specificata. Questa funzione deve sempre essere chiamata quando si alloca una variabile di tipo `NPVariant` e quando essa viene ritornata dal browser.

`NPN_GetStringIdentifier()...` Ritorna un identificatore *unico* per la stringa passata.

`NPN_IntFromIdentifier()...` Ritorna l'intero corrispondente all'identificatore passato. Esistono altre funzioni che consentono di ritornare gli altri tipi.

[22] `NPN_GetProperty()`, `NPN_HasProperty()`, `NPN_Invoke()`, `NPN_InvokeDefault()`, `NPN_RemoveProperty()`, `NPN_SetProperty()`

Capitolo 2

Linee guida per lo sviluppo di un plugin

Come detto in precedenza le funzioni delle NPAPI sono implementate in linguaggio C, mentre gli esempi dei plugin che fornisce mozilla[1] sono scritti in C++. Mozilla in questi esempi fornisce Makefiles (oppure progetti per la compilazione in Microsoft Visual Studio) che richiedono la presenza dei sorgenti di Firefox oppurtunamente configurati[2]. Inoltre nella sottodirectory common/ sono forniti alcuni files che implementano le varie funzioni delle NPAPI.[3] Questa implementazione delle funzioni NPP_* presuppone che l'istanza del plugin sia implementata attraverso una classe C++. Nella documentazione di Mozilla non vi è nessun cenno a questo tipo di implementazione, rendendo tutta la documentazione[4] abbastanza obsoleta in quanto le NPAPI vengono surclassate da funzioni membro della classe. Sebbene in teoria l'utilizzo di classi dovrebbe rendere il codice più strutturato e leggibile, in questa circostanza le NPAPI originarie risultano di per se molto chiare e relativamente semplici da utilizzare. Pertanto in tutti gli esempi seguenti e nello sviluppo del plugin app-wrapper non farò uso di classi C++.

Per semplificare e rendere più chiaro lo sviluppo di un plugin per Firefox, ho introdotto alcune macro che evitano di implementare funzioni che non hanno alcun utilizzo in plugin con determinate caratteristiche. Inoltre ho fornito un Makefile ben documentato che dovrebbe rendere la compilazione di plugin immediata.

[1] http://mxr.mozilla.org/mozilla-central/source/modules/plugin/sdk/samples/.
[2] cioè usati per compilare una versione completa di Mozilla Firefox.
[3] npp_gate.cpp, npn_gate.cpp e np_entry.cpp.
[4] https://developer.mozilla.org/en/Gecko_Plugin_API_Reference

2.1 I files utilizzati

L'organizzazione dei files forniti è la seguente:

La directory principale contiene il codice del plugin `app-wrapper` e alcuni files che contengono le linee guida per lo sviluppo di un plugin per Firefox, oltre alla documentazione;

La directory `include/` contiene i file header forniti da mozilla

La directory `examples/` contiene il codice degli esempi utilizzati per lo sviluppo di un plugin;

La directory `sdlEx/` contiene gli esempi riguardanti la libreria SDL e i problemi relativi all' *embedding* di applicazioni SDL in un plugin;

La directory `windows/` contiene una versione preliminare del plugin `app-wrapper` sviluppata per il sistema operativo `Microsoft Windows`.

Il comando `make` nella directory principale effettua la compilazione del plugin `app-wrapper`. L'esecuzione del comando `make` nelle sottodirectory `examples/` e `sdlEx/` compila e installa i vari esempi. Inoltre è possibile accedere ai vari esempi visualizzando le pagine `sdlEx/showSDLExample.html` e `examples/showExamples.html`.

2.1.1 Files necessari per lo sviluppo di un plugin

Come detto in precedenza, utilizzando le linee guida fornite non è necessario appoggiarsi a una classe C++ per la creazione di un plugin. I files necessari per la creazione di un plugin sono i seguenti:

- `myplugin.h` e `myplugin.c`: il file header è il principale file da includere per scrivere un plugin. Esso include a sua volta i file forniti da Mozilla, e adatta alcune funzioni a seconda della piattaforma in uso.[5] Contiene l'implementazione delle funzioni dell' `NPAPI` che lo sviluppatore non intende utilizzare[6] in modo tale da semplificare lo sviluppo del plugin e mette a disposizione funzioni per il debug completo. Implementa, se necessario, le funzione di inizializzazione e chiusura se l'utente non ne usa di personalizzate.

[5] Ad esempio il prototipo della funzione si inizializzazione è diversa se si sta usando un sistema operativo `Unix` o `Microsoft Windows`.

[6] Se veda più avanti il `Makefile` e le macro supportate.

- `npn.c`: contiene l'implementazione delle funzioni `NPN_*`. Queste implementazioni riprendono quelle utilizzate da Mozilla[7] senza utilizzare la notazione delle classi e con l'aggiunta di informazioni per il debug.
- La directory `include/` contiene i file header forniti da Mozilla e necessari per la compilazione del plugin.
- Il `Makefile`: per la compilazione del plugin è fortemente raccomandato l'utilizzo di un `Makefile`. É possibile adattare quello fornito per il plugin `app-wrapper` oppure utilizzare quelli presenti nella directory `examples/`. É importante ricordarsi di definire tutte le macro necessarie all'interno del `Makefile` altrimenti si perdono le funzionalità di questa implementazione.

2.2 Le macro introdotte

Per facilitare lo sviluppo di un plugin e per permettere allo sviluppatore di scrivere solo codice necessario allo scopo del plugin, ho introdotto alcune macro. Queste macro è necessario definirle nel `Makefile` in modo tale da renderle disponibile a tutti file. Le macro introdotte sono le seguenti:

- `PLUGINDEBUG`: indica che si vuole compilare il plugin con il supporto al debug completo. In questo modo vengono stampati messaggi di debug ad ogni invocazione di una funzione delle `NPAPI`, e quando è stato ritenuto oppurtuno. Lo sviluppatore può utilizzare la funzione `debMes(...)` definita nel file `myplugin.c` con la stessa sintassi della funzione `printf(...)` in modo tale da creare informazioni di debug. I messaggi vengono stampati su terminale (in Linux/FreeBSD) o su file (Windows).
- `PLUGINOUT`: utilizzata solo in Microsoft Windows, e serve per settare il percorso e il nome del file per il debug. Infatti si Windows non esiste lo *stdout* e quindi le informazioni di debug sono salvate su un file. É possibile vedere queste informazioni in tempo reale aprendo un terminale (o una shell `DOS`) e digitare un comando del tipo `tail -f nome del file`.
- `NPN_MACROS`: implementa le funzioni `NPN_*` come macro. In questa maniera il codice risulterà più veloce, a scapito della mancanza di codice di debug e soprattutto della mancanza di controlli sui parametri passati alle funzioni. Il comportamento del browser risulta non definito se vengono passati parametri di tipo sbagliato a tali funzioni. Si consiglia di non utilizzare questa macro finchè il codice non è stato testato approfonditamente.

[7] Nel file `npn_gate.cpp`.

- `USEINITIAL`: se questa macro è definita, allora lo sviluppatore si appoggia alle funzioni di inizializzazione e chiusura[8] predefinite. Spesso è consigliato definire questa macro, in quanto generalmente è necessario utilizzare funzioni di inizializzazione personalizzate quando si deve allocare della memoria condivisa fra più istanze del plugin.

- `WANT_X11`: se la macro è definita viene detto a Mozilla che si intende utilizzare la piattaforma X11. In generale in caso di plugin per Linux/FreeBSD è necessario definire questa macro anche se non si usano esplicitamente funzioni di `X11` in quanto altrimenti non si più accedere ad alcuni membri privati riguardanti l'implementazione della finestra.

- `WANT_SDL`: informa il plugin che si vuole utilizzare la libreria SDL e rende disponibile la funzione `HackSDL()` che setta la variabile d'ambiente `SDL_WINDOWID` con il valore della finestra assegnata al plugin in modo tale che un'applicazione SDL venga eseguita come plugin.

- `WANT_STREAM`: Se la macro è definita significa che lo sviluppatore sta creando un plugin che necessita delle funzioni riguardanti lo streaming. In questo caso è necessario definire nel proprio codice tutte le funzioni riguardanti lo streaming. Nel caso lo sviluppatore non intende utilizzare queste funzioni, la macro non deve essere definita, cosicchè saranno utilizzate le implementazioni *stub* presenti nel file `myplugin.c`.

2.3 Linee guida

Il file `README.start` fornisce le informazioni primarie per poter scrivere un plugin per il browser Firefox. La prima cosa da fare è includere il file `myplugin.h` nel proprio codice sorgente. Questo file include tutti i file necessari per la piattaforma in uso, e fornisce alcune funzioni utili per il debug. A seconda delle funzionalità richieste è necessario dichiarare alcune macro (si veda il paragrafo precedente) nel `Makefile` in modo tale che vengano incluse o meno alcune funzioni predefinite.

In linea di massima le funzioni facenti parte delle `NPAPI` che un programmatore deve necessariamente implementare sono le seguenti:

- `NPP_New()` e `NPP_Destroy()`: necessarie per la creazione e la distruzione di una istanza del plugin. Nella funzione di creazione è necessario allocare la memoria riservata all'istanza del plugin, che sarà deallocata nella funzione di distruzione.

[8] `NP_Initialize()` e `NP_Shutdown()`

- `NPP_GetValue()`: è necessario specificare in questa funzione una descrizione del plugin, e le caratteristiche che si vuole richiedere di usare (ad esempio se si sta scrivendo un plugin *scriptabile*).
- `NPP_SetWindow()`: è la funzione che si occupa di disegnare il plugin nella finestra assegnatagli dal browser.
- `NPP_GetMIMEDescription()`: indica i MIME types che il plugin supporta. É utilizzata solo in ambiente Unix.

Inoltre è possibile implementare le funzioni di inizializzazione e chiusura personalizzate[9] non dichiarando la macro `USEINITIAL`.

In genere si consiglia di memorizzare tutte le informazioni private di un istanza in una struttura `_instance`, che sarà memorizzata nel campo `pdata` passato nel parametro `NPP` di ogni funzione delle `NPAPI`.

Gli esempi nel capitolo successivo mostreranno in maniera più esauriente come scrivere un plugin per sistemi operativi *Unix-like*. Nell'ultimo capitolo verranno introdotte le differenze principale per scrivere un plugin compatibile con `Microsoft Windows`.

[9] attraverso puntatori a funzione, si veda gli esempi del capitolo successivo.

Capitolo 3

Descrizione del funzionamento di un plugin attraverso esempi

In questo capitolo tratteremo lo sviluppo di plugin attraverso esempi via via sempre più complessi. I primi esempi mostreranno i passi fondamentali che uno sviluppatore dovrebbe seguire per scrivere un plugin, successivamente verrà trattato il caso di plugin scriptabili.

3.1 Le basi

In questo paragrafo vedremo lo sviluppo di un plugin semplice che non ha nessuna utilità tranne mostrare allo sviluppatore come iniziare a scrivere un plugin. In tutti gli esempi è presente molto codice di debug; perciò è altamente consigliato avviare il browser da un terminale in modo tale da poter leggere l'output dal terminale stesso. Lo sviluppo sarà effettuato seguendo le linee guida esposte nel capitolo precedente.

Listing 3.1: *La struttura dati _instance per il plugin1*

```
struct _instance {
        int mX, mY; /* Left high corner coordinates of the window */
        int mWidth, mHeight; /* Plugin window size */
        Display *mDisplay;
        Window mWindow; /* Window id */
};
```

Listing 3.2: *Spezzone della funzione NPP_New() per il plugin1*

```
NPError NPP_New(NPMIMEType pluginType, NPP instance,
                uint16 mode, int16 argc, char *argn[],
                char *argv[], NPSavedData *saved) {
        struct _instance * my_instance;
        my_instance = NPN_MemAlloc(sizeof(struct _instance));
        bzero(my_instance, sizeof(my_instance));
        instance->pdata = (void*) my_instance;
        return NPERR_NO_ERROR;
}
```

Listing 3.3: *La funzione NP_GetMIMEDescription() per il plugin1*

```
char *NP_GetMIMEDescription(void) {
        debMes("calling %s\n", __FUNCTION__);
        return "application/x-e1:e1:Plugin1";
}
```

3.1.1 Struttura dati usata

La struttura dati privata del plugin (codice 3.1) contiene soltanto alcune informazioni geometriche sulla finestra, un puntatore al Display del server X usato e un identificatore della finestra del plugin. Non sono presenti altre variabili, in quanto come già detto in precedenza questo plugin non ha nessuna funzione utile.

3.1.2 Inizializzazione e distruzione

Sono state usate le funzioni predefinite per l'inizializzazione e chiusura globali. Per questo plugin infatti non è necessario allocare spazio condiviso fra le varie istanze. La prima[1] funzione chiamata è `NP_GetMIMEDescription()` (codice 3.3), che indica al browser il MIME type supportato da questo plugin, e l'eventuale estensione del

[1] In realtà come già detto in precedenza questa funzione è chiamata soltanto quando il browser si accorge che il codice del plugin è stato modificato.

Listing 3.4: *Spezzone della funzione NP_Destroy() per il plugin1*

```
NPError NPP_Destroy(NPP instance, NPSavedData **save) {
        NPN_MemFree(instance->pdata);
        return NPERR_NO_ERROR;
}
```

Listing 3.5: *Spezzone della funzione `NPP_GetValue()` per il plugin1*

```
NPError NPP_GetValue(NPP instance, NPPVariable variable, void *value) {
        switch (variable) {
        case NPPVpluginNameString:
                debMes("  %s: requested %d(NPPVpluginNameString)\n",
                        __FUNCTION__, variable);
                *((char **)value) = "plugin1";
                break;
        case NPPVpluginDescriptionString:
                debMes("  %s: requested %d(NPPVpluginDescriptionString)\n",
                        __FUNCTION__, variable);
                *((char **)value) = "plugin1 - This is a test example plugin";
                break;
        return NPERR_NO_ERROR;
}
```

file che lo invoca. In questo caso il MIME type supportato è `application/x-e1`, l'estensione associate è `.e1` e il nome del plugin è `Plugin1`.

Per quanto riguarda la creazione dell'istanza del plugin (codice 3.2) per prima cosa viene allocato lo spazio necessario per memorizzare la struttura `_instance` (codice 3.2 linea 5), poi ne viene azzerato il contenuto e successivamente viene associato il puntatore a questa struttura alla nuova istanza `NPP` passata dalla funzione `NPP_New()` (codice 3.2 linea 7). Infine se non ci sono stati errori[2] la funzione ritorna `NPERR_NO_ERROR`.

Dopo la creazione dell'istanza (e anche in momenti successivi) il browser può chiamare la funzione `NPP_GetValue()` (codice 3.5) per chiedere informazioni al plugin. Generalmente si specifica qui il nome e una descrizione completa del plugin, e in più si può informare il browser delle caratteristiche che vogliamo usare (ad esempio se vogliare creare un plugin scriptabile ...). Tutti i casi usabili in questa funzione sono specificati nel codice sorgente del plugin.

La distruzione dell'istanza è effettuata dalla funzione `NPP_Destroy()` (codice 3.4). In questo esempio viene deallocata la memoria allocata dalla funzione `NPP_New()`. Come consigliato da mozilla l'allocazione e la deallocazione della memoria è effettuata tramite le funzioni fornite da NPAPI[3].

3.1.3 Funzionamento

In questo esempio la funzionalità del plugin è svolta interamente dalla funzione `NPP_SetWindow()`. Questa funzione è chiamata la prima volta dal browser nel momento in cui crea la finestra assegnata al plugin, e le volte successive quando cambiano le geometrie della finestra o quando è necessario effettuare un aggiornamento del

[2] In questo spezzone di codice sono stati omessi alcuni test di verifica.

[3] `NPN_MemAlloc()` e `NPN_MemFree()`.

Listing 3.6: *Spezzone della funzione `NPP_SetWindow()` per il plugin1*

```
1  NPError NPP_SetWindow(NPP instance, NPWindow* window) {
2          struct _instance * my_instance;
3          my_instance = (struct _instance *) instance->pdata;
4          if (my_instance->mWindow == (Window) window->window) {
5                  if ((my_instance->mWidth != window->width) || [...] ) {
6                          /* Stampo le coordinate della finestra */
7                  }
8                  else {
9                          debMes("  %s: repaint the window\n", __FUNCTION__);
10                 }
11         } else {
12                 debMes("  %s: first time in the function\n", __FUNCTION__);
13                 my_instance->mWindow = (Window) window->window;
14                 NPSetWindowCallbackStruct *ws_info =
15                         (NPSetWindowCallbackStruct *)window->ws_info;
16                 my_instance->mDisplay = ws_info->display;
17         }
18         my_instance->mX = window->x;
19         [...]
20         return NPERR_NO_ERROR;
21 }
```

disegno. Per prima cosa viene estratta la struttura che contiene i dati privati del plugin dal parametro `instance` passato dalla funzione. Poi viene confrontato l'identificatore della finestra passata dalla funzione attraverso il parametro `window` con quello memorizzato in precedenza (codice 3.6 linea 4). Se i due identificatori sono uguali significa che non è la prima volta che questa funzione è eseguita, e viene effettuato un test per vedere se è cambiata la geometria della finestra[4] oppure se è solo necessario effettuare un refresh della finestra stessa. Se invece i due identificatori non sono uguali, si può supporre[5] che la funzione `NPP_SetWindow()` è stata chiamata per la prima volta. In questo caso si memorizza l'dentificatore della finestra passatoci dalla funzione nel campo privato dell'istanza, e si estraggono alcune informazioni sulla finestra dipendenti dalla piattaforma in uso dalla struttura `NPSetWindowCallbackStruct` (codice 3.6 linea 14). A questo punto è possibile inizializzare il sottosistema grafico a seconda della piattaforma usata. Infine (codice 3.6 linea 18) si memorizzano tutte le informazioni relative alla posizione della finestra nei capi privati del plugin. A questo punto si potrebbero chiamare le funzioni per effettuare eventuali disegni sulla finestra.

Un tipico output dell'esecuzione del plugin è mostrato nel codice 3.7. Sono mo-

[4]In questo esempio non è possibile cambiare le dimensioni della finestra; le uniche coordinate che possono cambiare sono la posizione di inizio della finestra che cambiano se la finestra del browser viene ridimensionata.

[5]Teoricamente potrebbe essere passato un window id diverso da quello passato in precedenza. Ciò significa che la finestra del plugin è stata distrutta e ne è stata creata una nuova. In questo caso è necessario effettuare nuovamente le inizializzazioni come la prima volta

Listing 3.7: *Output su terminale del plugin1*

```
calling NP_Initialize
-- NPN_Version called
Plugin version: 0.17    Browser support: 0.19
calling NPP_New
-- NPN_MemAlloc called
calling NPP_GetValue
  NPP_GetValue: requested 14(NPPVpluginNeedsXEmbed)
calling NPP_SetWindow
  NPP_SetWindow: first time in the function
calling NPP_GetValue
  NPP_GetValue: requested 15(NPPVpluginScriptableNPObject)
calling NPP_GetValue
  NPP_GetValue: requested 11(NPPVpluginScriptableIID)
calling NPP_SetWindow
  NPP_SetWindow: repaint the window
calling NPP_SetWindow
  NPP_SetWindow: Old size 240x200, old position 8x198; New size 240x200, new
  position 8x236
calling NPP_SetWindow
  NPP_SetWindow: repaint the window
calling NPP_SetWindow
  NPP_SetWindow: repaint the window
calling NPP_SetWindow
  NPP_SetWindow: repaint the window
calling NPP_SetWindow
  NPP_SetWindow: repaint the window
calling NPP_Destroy
-- NPN_MemFree called
calling NP_Shutdown
```

strate tutte le funzioni chiamate sia dal browser che dal plugin. Da notare le chiamate del browser alla funzione `NPP_GetValue()` per chiedere infomazioni sul plugin (in questo caso è chiesto se il plugin richede *XEmbed*, se è scriptabile attraverso *nprunti-me* o attraverso modi obsoleti). La funzione `NP_GetMIMEDescription()` non è stata chiamata in quanto questo plugin era già stato eseguito in precedenza. .

3.1.4 Compilazione

Per compilare il plugin è necessario dare il comando `make plugin1` che invocherà il corretto *makefile* per la compilazione. Le macro usate sono `-DPLUGINDEBUG` per il debug e `-DUSEINITIAL` in quanto in questo primo esempio utilizzeremo le funzioni di inizializzazione e chiusura predefinite. L'installazione avverrà in automatico nella directory `~/.mozilla/plugins`, in modo tale che il plugin risulti attivo per il singolo utente.

Listing 3.8: *Modifiche al plugin1 per utilizzare funzioni di inizializzazione e shutdown personalizzate.*

```
1  NPError my_Initialize(NPNetscapeFuncs* npn, NPPluginFuncs* npp) {
2          debMes("calling %s\n", __FUNCTION__);
3          debMes("  Using my own initialize function\n");
4          return NPERR_NO_ERROR;
5  }
6
7  NPError my_Shutdown(void) {
8          debMes("calling %s\n", __FUNCTION__);
9          debMes("  Using my own shutdown function\n");
10         return NPERR_NO_ERROR;
11 }
```

3.1.5 Uso di funzioni di inizializzazione personalizzate

Il successivo esempio (file `plugin1a`) differisce dal precedente soltanto nell'utilizzo di funzioni personalizzate di inizializzazione e chiusura.

Le due funzioni di inizializzazione (`my_Initialize()`) e shutdown (`my_Shutdown()`) non fanno niente di diverso, mostrano soltanto un messaggio sul terminale nel momento in cui vengono chiamate (codice 3.8). É necessario commentare nel *Makefile* la macro `-DUSEINITIAL` in modo tale che i due puntatori alle funzioni personalizzate vengano inizializzati correttamente nella funzione di inizializzazione predefinita.

3.2 I plugin scriptabili

In questo paragrafo tratteremo due esempi riguardanti lo scripting dei plugin. Con il termine scripting si intende la possibilità di chiamare funzioni Javascript (o accedere al contenuto del DOM) dall'interno del codice del plugin, e esportare funzioni (e variabili) del plugin rendendole chiamabili da Javascript.

3.2.1 L'esempio plugins1

Il primo esempio mostra come è possibile accedere agli elementi del DOM dall'interno del plugin. Al momento della creazione dell'istanza il plugin recupera alcune informazioni riguardanti il DOM e le mostra su terminale. Inoltre alla prima esecuzione della funzione `NPP_SetWindow()` visualizza una finestra di conferma nel browser, e il risultato della risposta (un boolean) verrà mostrato su terminale.

Listing 3.9: *Spezzone della funzione `NPP_New()` per plugins1.*

```
[...]
if (!plugin_calls_js(instance, "document.location.href", &res))
        debMes("  %s: Error retreiving href\n", __FUNCTION__);
else {
        debMes("  %s: href=%s\n", __FUNCTION__,
                  NPVARIANT_TO_STRING(res).utf8characters);
        /* Now we must release the object used for retreive DOM infos */
        NPN_ReleaseVariantValue(&res);
}
[...]
/* Altre chiamate a plugin_calls_js(...)*/
[...]
```

Listing 3.10: *Spezzone della funzione `NPP_SetWindow()` per plugins1.*

```
[...]
if (!plugin_calls_js(instance,"window.confirm
        ('Hello!This confirm window has been create by plugin')",&res))
        debMes("  %s: Error calling window.confirm\n", __FUNCTION__);
else {
        my_instance->result = NPVARIANT_TO_BOOLEAN(res);
        if (my_instance->result == TRUE) {
                debMes("  %s:javascript return true\n", __FUNCTION__);
        }
        else {
                debMes("  %s:javascript return false\n",  __FUNCTION__);
        }
        NPN_ReleaseVariantValue(&res);
}
[...]
```

Struttura dati usata

La struttura dati usata è molto simile all'esempio precedente, con la sola aggiunta di un campo boolean che contiene il risultato della chiamata *window.confirm*.

Funzionamento

Sono state utilizzate le funzioni di inizializzazione e shutdown predefinite in quanto sufficienti per lo scopo di questo esempio. La prima parte della funzione di creazione dell'istanza (`NPP_New()`) è simile al precedente esempio. Dopo l'allocazione della memoria e le inizializzazioni alla struttura dati, vengono effettuate quattro chiamate alla funzione `plugin_calls_js()` (codice 3.9) per eseguire codice Javascript dall'interno del plugin effettuando chiamate a funzioni facenti parte delle *NPAPI* (vedi appendice A.2).

Listing 3.11: *Frammenti di output per plugins1.*

```
1  calling NPP_New
2  -- NPN_MemAlloc called
3  [...]
4  calling plugin_calls_js
5  -- NPN_GetValue called
6  -- NPN_GetValue returns 0
7  -- NPN_Evaluate called
8  -- NPN_Evaluate returns 1
9  -- NPN_ReleaseObject called; counter is 1
10 -- NPN_ReleaseObject returns; counter is 0
11   NPP_New: userAgent=Mozilla/5.0 (X11; U; Linux x86_64; it; rv:1.9.0.9) Gecko
     /2009042311 Gentoo Firefox/3.0.9
12 [...]
13   NPP_SetWindow: first time in the function
14 calling plugin_calls_js
15 -- NPN_GetValue called
16 -- NPN_GetValue returns 0
17 -- NPN_Evaluate called
18 -- NPN_Evaluate returns 1
19 -- NPN_ReleaseObject called; counter is 1
20 -- NPN_ReleaseObject returns; counter is 0
21   NPP_SetWindow:javascript return true
22 -- NPN_ReleaseVariantValue called
23 [...]
```

La prima volta che viene chiamata la funzione `NPP_SetWindow()` appare una finestra di conferma nella quale è mostrato un messaggio generato dal plugin. Il risultato di questa scelta è convertito in boolean (attraverso la macro `NPVARIANT_TO_BOOLEAN()`) e memorizzato nella variabile privata `result`. Da notare che ogni volta che viene chiamata la funzione `plugin_calls_js()` è necessario chiamare esplicitamente la funzione `NPN_ReleaseVariantValue(&res)` per liberare l'oggetto `NPVariant` passatoci dal browser.

Un tipico output su terminale è mostrato nel codice 3.11. Alla linea 4 si vede la chiamata alla funzione `plugin_calls_js` con la richiesta di informazioni sull' *user agent* del browser. Tale funzione chiama a sua volta le funzioni `NPN_GetValue()`, `NPN_Evaluate()` e `NPN_ReleaseObject()`, e infine è stampato il risultato della chiamata. Alla riga 14 è mostrata la chiamata Javascript effettuata all'interno della funzione `NPP_SetWindow()`, con il risultato del click dell'utente[6].

3.2.2 L'esempio plugins2

L'esempio succesivo tratta la seconda parte dei plugin scriptabili: rendere accessibile all'esterno del plugin funzioni e variabili interne. Per far ciò è necessario creare un

[6] OK=true, Cancel=false.

Listing 3.12: *Struttura dati per plugins2.*

```
1  struct _my_obj {
2          NPObject me; /* the actual object. */
3          struct _instance * instance; /* Pointer to te actual plugin instance */
4          NPIdentifier func1; /* [my_fn1] The function that will be accessible
           from javascript */
5          NPIdentifier valId; /* [value] NPVariant representing the integer val */
6  };
7  struct _instance {
8          long mWindow; /* Window id */
9          int mX, mY; /* Left high corner coordinates of the window */
10         int mWidth, mHeight; /* Window dimension */
11         Display *mDisplay;
12         struct _my_obj *objS; /* A pointer to the scriptable object*/
13         int val; /* The value set and read through javascript */
14 };
```

oggetto scriptabile (`NPObject`) e renderlo disponibile al browser. Questo esempio esporta una funzione e una variabile intera. Premendo il pulsante *Run* viene eseguita la funzione che mostra un messaggio, mentre con i pulsanti *Set value* e *Get value* si setta e si legge un valore intero.

Struttura dati usata

La struttura dati usata è mostrata nel codice 3.12. La struttura `_instance` è stata ampliata con un intero `val` che rappresenta la variabile esportata dal plugin, e con un puntatore a una struttura `_my_obj` che rappresenta l'oggetto scriptabile esportato al browser. All'oggetto standard `NPObject` è stato sovrapposto un oggetto personalizzato `_my_obj` in modo tale da poter memorizzare ulteriori informazioni nell'oggetto scriptabile senza dover utilizzare variabili globali. All'interno di questo oggetto è presente un puntatore all'istanza del plugin in modo tale da poter accedere a tutti i dati del plugin, e i due identificatori[7] della funzione e della variabile esportati.

Inizializzazione

Le funzioni `NPP_New()` e `NPP_SetWindow()` sono simili all'esempio base. Nella prima viene allocato lo spazio necessario per memorizzare l'istanza e vengono inizializzati i vari membri, mentre la seconda stampa soltanto messaggi di informazione e di debug su terminale.

La funzione `NPP_GetValue()` (codice 3.13) è stata ampliata in modo tale da informare il browser che il plugin crea ed esporta un oggetto scriptabile. Il *caso*

[7] di tipo `NPIdentifier`

Listing 3.13: *Parte della funzione `NPP_GetValue()` per invocare la creazione dell'oggetto scriptabile, per plugins2.*

```
1  [...]
2  case NPPVpluginScriptableNPObject:
3          debMes("  %s: requested %d(NPPVpluginScriptableNPObject)\n",
4                  __FUNCTION__, variable);
5          *(NPObject **)value = my_GetScriptableObject(instance);
6          break;
7  [...]
```

Listing 3.14: *La funzione `my_GetScriptableObject()` per la creazione dell'oggetto scriptabile, per plugins2.*

```
1  NPObject *my_GetScriptableObject (NPP instance) {
2  [...]
3          if (!my_instance->objS) {
4                  debMes("  %s: creating new object\n", __FUNCTION__);
5                  my_instance->objS = (struct _my_obj*)
6                                  NPN_CreateObject(instance, &xyz_scriptable);
7                  if (my_instance->objS) {
8                          my_instance->objS->func1 =
9                                  NPN_GetStringIdentifier("my_fn1");
10                         my_instance->objS->valId =
11                                 NPN_GetStringIdentifier("value");
12                 }
13                 else
14                    debMes("  %s: Error creating new object\n", __FUNCTION__);
15         } else {
16                 debMes("  %s: object already existing\n", __FUNCTION__);
17                 NPN_RetainObject ((NPObject*)my_instance->objS);
18         }
19         return (NPObject*)(my_instance->objS);
20 }
```

`NPPVpluginScriptableNPObject` rappresenta l'uso della scriptabilità attraverso l'uso di oggetti `NPObject`. Quando il browser fa questa richiesta, viene chiamata la funzione `my_GetScriptableObject()` che crea l'oggetto, e lo ritorna nella variabile `value` che viene passata al browser.

L'oggetto scriptabile

La funzione che effettua la creazione dell'oggetto scriptabile è `my_GetScriptableObject()`. Questa funzione controlla se l'oggetto è già stato creato in precedenza (codice 3.14 linea 3) e in caso affermativo[8] incrementa il

[8] Il browser può invocare la funzione `NPP_GetValue()` più volte durante l'esecuzione. In questi casi è necessario non creare pià copie dell'oggetto, ma indicare che l'oggetto è stato referenziato più volte.

contatore degli oggetti referenziati attraverso la funzione `NPN_RetainObject`(linea 17). Con la chiamata alla funzione `NPN_CreateObject()` viene effettuata la creazione dell'oggetto scriptabile. Questa funzione ha come parametri l'istanza del plugin, e una struttura `NPClass` che rappresenta l'implementazione dell'oggetto. Successivamente (linea 9) viene associata all'identificatore `my_fn1` la chiamata alla funzione `func1()` e all'identificatore `value` la variabile `valId`. Questi saranno i due identificatori per accedere alla funzione e alla variabile dall'esterno del plugin.

L'implementazione della NPClass

Poichè abbiamo usato un oggetto personalizzato, è necessario implementare le funzioni `_allocate()` e `_deallocate()` per allocare e deallocare lo spazio necessario per memorizzare l'oggetto. Nell'allocazione è anche necessario inizializzare il puntatore all'istanza del plugin, in quanto l'istanza non sarà accessibile dalle altre funzioni della classe. Le seguenti funzioni della NPCLASS sono state personalizzate in modo da informare il browser delle funzioni e delle variabili esportate:

la funzione `_hasProperty()` dice al browser se un identificatore è un membro dell'oggetto, in questo caso esiste solo la variabile `valId`;

la funzione `_getProperty()` ritorna nel parametro `Result` il valore della variabile intera convertito nel tipo `NPVariant`;

la funzione `_setProperty()` setta la variabile intera con il valore (convertito prima in intero) del parametro `value`;

la funzione `_hasMethod()` informa il browser se l'indentificatore `name` è una funzione dell'oggetto;

la funzione `_invoke()` invoca la funzione `name` con gli eventuali parametri (in questo caso nessun parametro è passato alla funzione); il risultato della funzione è passato al browser attraverso il parametro `result`.

In tutte queste funzioni di accesso ai metodi e ai membri dell'oggetto è sempre per prima cosa verificato il nome del membro o metodo da eseguire; se questo nome non combacia con quelli implementati nel plugin viene ritornato il valore `false`.

Le altre funzioni della NPClass sono state implementate come *stub* in quanto non necessarie per questo plugin.

Sarà poi il browser a *rilasciare* l'ggetto quando necessario.

Listing 3.15: *L'implementazione della NPClass per l'oggetto scriptabile per plugins2.*

```
1  static NPObject *_allocate(NPP instance, NPClass *cl) {
2          struct _my_obj * o = malloc(sizeof(struct _my_obj));
3          o->instance = (struct _instance *) instance->pdata;
4          return (NPObject *)o;
5  }
6
7  static void _deallocate(NPObject *o) {
8          struct _my_obj * obj = (struct _my_obj *) o;
9          obj->instance->objS = NULL;
10         free(o);
11 }
12
13 static bool _hasProperty (NPObject *o, NPIdentifier name) {
14         struct _my_obj *x = (struct _my_obj *)o;
15         return (name == x->valId);
16 }
17
18 static bool _getProperty (NPObject *o, NPIdentifier name, NPVariant *result) {
19         struct _my_obj *x = (struct _my_obj *)o;
20         if (name == x->valId) {
21                 INT32_TO_NPVARIANT(x->instance->val, *result);
22                 return true;
23         }
24         return false;
25 }
26
27 static bool _setProperty (NPObject *o, NPIdentifier name, const NPVariant *value
   ) {
28         struct _my_obj *x = (struct _my_obj *)o;
29         if (name == x->valId) {
30                 x->instance->val = NPVARIANT_TO_INT32(*value);
31                 return true;
32         }
33         return false;
34 }
35
36 static bool _hasMethod(NPObject *o, NPIdentifier name) {
37         struct _my_obj *x = (struct _my_obj *)o;
38         return (name == x->func1);
39 }
40
41 static bool _invoke (NPObject *o, NPIdentifier name,
42         const NPVariant *argv, uint32_t argc, NPVariant *result) {
43         struct _my_obj *x = (struct _my_obj *)o;
44         if (name == x->func1) {
45                 STRINGZ_TO_NPVARIANT(strdup("This string is created by plugin!")
                  ,
46                                                 *result);
47                 return TRUE;
48         }
49         return FALSE;
50 }
```

Listing 3.16: *Il codice HTML per l'esempio plugins2.*

```
[...]
<embed type="application/x-s2" id="test"></embed>
<br/><br/>
<form>
   <input type="button" value="Run" onclick=alert(test.my_fn1())><br>
   <input type="button" value="Set value" onclick=setValue()><br>
   <input type="button" value="Get value" onclick=getValue()><br>
</form>
<script>
function setValue() {
        var res = window.prompt("Insert an integer to pass to plugin");
        var i = parseInt(res);
        test.value = i;
}
function getValue() {
        var res = test.value;
        window.alert("The integer from the plugin is: " + res);
}
</script>
[...]
```

Il codice `HTML` per l'esempio

Il codice `HTML` per visualizzare l'esempio è mostrato nel listato 3.16. Si può notare che il plugin è invocato attraverso il tag `embed`, al quale è stato assegnato un id uguale a `test`, con il quale è possibile accedere alle funzione e ai membri del plugin. Premendo il pulsante *Run* viene eeguita la funzione `test.my_fn1()` e visualizzato il risultato attraverso una finestra Javascript di *alert*. Premendo i pulsanti *Set value* o *Get value* viene settata o visualizzato un intero, accedendo in scrittura o lettura al membro `value` del plugin.

Capitolo 4

Le applicazioni SDL: esempi di integrazione in un plugin

SDL[1] è una libreria multipiattaforma scritta in C rilasciata sotto la licenza GNU LGPL[2] per lo sviluppo di applicazioni multimediali. SDL crea un livello astratto al di sopra di varie piattaforme software grafiche e sonore e dunque può controllare video, audio, CDROM e altre periferiche. Essendo multipiattaforma, è possibile creare applicazioni su un sistema operativo e portarlo in altri senza apportare modifiche importanti al codice. Le piattaforme supportare sono molte, a partire da Microsoft Windows, Linux/Unix, fino ad arrivare ai sistemi Windows CE e SymbianOS.

4.1 Breve introduzione a SDL

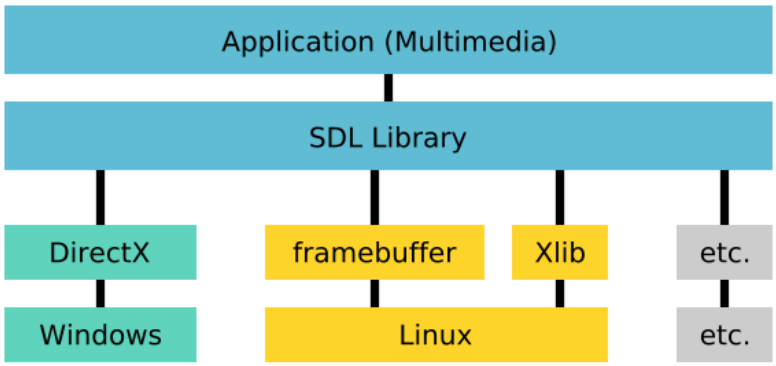
L'implementazione di SDL è relativamente semplice: è un wrapper leggero e multi piattaforma che fornisce il supporto alle operazioni 2D sui pixel, suoni, accesso ai file, gestione degli eventi, temporizzatori, thread e altro. É spesso usata come complemento alle OpenGL settando l'output grafico e fornendo la gestione dell'input del mouse e della tastiera, che sono ben oltre lo scopo delle OpenGL.

La libreria è suddivisa in parecchi sottosistemi, tra i quali il Video (gestisce sia le funzioni per le superfici e l'OpenGL), l'Audio, il CD-ROM, il Joystick e il sottosistema Timer. A parte il supporto basico a basso livello, vi sono alcune librerie ufficiali di supporto che forniscono funzionalità addizionali. Queste comprendono le librerie standard, sono fornite sul sito ufficiale e incluse nella documentazione ufficiale[3]:

SDL_image : supporto per diversi formati di immagini;

[1] http://www.libsdl.org/
[2] http://www.gnu.org/licenses/lgpl.html
[3] http://www.libsdl.org/cgi/docwiki.cgi

Figura 4.1: *Livello di astrazione della libreria SDL per le piattaforme più comuni.*



SDL_mixer : funzioni audio complesse, principalmente per il mixing dei suoni;

SDL_net : supporto alla rete;

SDL_ttf : supporto alla renderizzazione dei font TrueType;

SDL_rtf : renderizzazione semplice del Rich Text Format.

Come si nota dalla figura 4.1 su Microsoft Windows SDL fornisce un wrapper per le DirectX[4], mentre sulle piattaforme X11 SDL usa le Xlib per comunicare con il sistema grafico sottostante[5].

In questo capitolo vedremo attraverso alcuni esempi come è possibile inglobare un'applicazione SDL in un plugin per Firefox. Il problema risiede nel fatto che di default un'applicazione SDL crea una nuova finestra per l'esecuzione. Esiste la possibilità di settare la variabile d'ambiente `SDL_WINDOWID` con il valore di un *windowid* in modo da forzare l'esecuzione dell'applicazione in una finestra già esistente. In questo modo però l'applicazione SDL non sarà in grado di gestire nessun evento del mouse o della tastiera, né di ridimensionare la finestra.

Nei paragrafi successivi vedremo come sarà possibile risolvere questo problema, sia effettuando modifiche all'applicazione che modificando la libreria SDL. La trattazione riguarderà i sistemi operativi *UNIX like*; per una soluzione per i sistemi operativi Microsoft Windows si vedi il capitolo 6.

[4] a partire dalla versione 1.2 utilizza le DirectX 7.

[5] oppure il framebuffer se non è usato un server X.

4.1.1 Le finestre di SDL

Vediamo brevemente l'architetture delle finestre usate da un applicazione SDL, quando viene eseguita nella propria finestra. Esistono 3 finestre create da SDL:

- `WMwindow`: è la finestra che viene *gestita* dal *window manager* del sistema. É creata come finestra figlia della finestra *root* e non è utilizzata per effettuare il disegno dell'applicazione, ma come contenitore della finestra `SDL_Window` e per gestire alcuni eventi (come la gestione della tastiera, ...).[6]
- `SDL_Window`: è la finestra dove avviene il disegno dell'applicazione SDL. Questa finestra è creata come figlia della finestra `WMwindow` e gestisce gli eventi del mouse (pressione dei bottoni, movimento, ...)[7]
- `FSwindow`: è la finestra che si occupa di disegnare lo schermo quando si passa alla visualizzazione a schermo intero.

Quando si tenta di eseguire un'applicazione SDL all'interno di una finestra esistente[8], le due finestre `SDL_Window` e `WMwindow` coincidono, e la finestra `FSwindow` non viene creata. Inoltre non vengono effettuate le chiamate alla funzione `XSelectInput()` cosicchè tutti gli eventi non possono essere gestiti dall'applicazione. Vedremo nel seguito del capitolo una possibile soluzione a questo problema.

4.2 L'applicazione di esempio

L'applicazione SDL che useremo come esempio[9] è un semplice programma che disegna un rettangolo riempito con un gradiente dal nero a uno dei tre colori primari, dando all'utente la possibilità di spostarsi da un gradiente all'altro premendo le frecce su/giu e di ingrandire o ridurre il rettangolo premendo le frecce destra/sinistra. Inoltre è aggiunta la possibilità di passare alla visualizzazione a schermo intero e tornare alla modalità finestra premento semplicemente la barra spaziatrice. Per uscire dal programma è sufficiente premere `Esc` o fare click sulla x della finestra.

[6] La maschera degli eventi associata a questa finestra è: `FocusChangeMask | KeyPressMask | KeyReleaseMask | PropertyChangeMask | StructureNotifyMask | KeymapStateMask`.

[7] in particolare la maschera degli eventi perquesta finestra sono: `EnterWindowMask LeaveWindowMask | ButtonPressMask | ButtonReleaseMask | PointerMotionMask | ExposureMask`.

[8] settando la variabile d'ambiente `SDL_WINDOWID` con il valore di un *window id* valido

[9] si veda il file `sdlEx/sdlVanilla.c`

4.3 Lo sviluppo

Vedremo in questo paragrafo come è stato risolto il problema dell'embedding dell'applicazione SDL di esempio all'interno di un plugin.

4.3.1 Il problema

Per mostrare il problema useremo l'applicazione `sdlVanilla` descritta in precedenza e un plugin (`plugin2`) che agisce da contenitore per l'applicazione.

Il plugin

Il plugin utilizzato per questo primo esempio è `sdlEx/plugin2`. Il plugin crea una finestra vuota nella quale viene eseguita l'applicazione il cui percorso è indicato nell'attributo `progtorun` passato dal codice `HTML`. Nella funzione `NPP_New()` viene controllato se esiste l'attributo `progtorun` che conterrà l'applicazione SDL da eseguire all'interno del plugin. Durante la creazione della finestra vuota, nella funzione `NPP_SetWindow()`, viene estratto il window id ed effettuato l'hack di SDL (cioè viene settata la variabile d'ambiente `SDL_WINDOWID` con il window id della finestra esistente), dopodichè viene effettuata una fork ed eseguita l'applicazione SDL.

Esecuzione dell'applicazione `sdlVanilla`

Se proviamo ad eseguire l'applicazione `sdlVanilla` all'interno del plugin, invocando il codice HTML mostrato nel listato 4.1, si nota che il rettangolo viene visualizzato all'interno della finestra assegnata al plugin.

Listing 4.1: *Codice HTML per eseguire l'applicazione `sdlVanilla` all'interno del `plugin2`.*

```
<embed type="application/x-p2" progtorun="./sdlVanilla"></embed>
```

La pressione dei tasti o del mouse non ha nessun effetto, e non è possibile cambiare colore allo sfondo ne ridimensionare la finestra. Inoltre si può notare che sovrapponendo una qualsiasi finestra al rettangolo, questo non viene successivamente più ridisegnato.[10] Questo significa che l'applicazione SDL non riesce a gestire nessun tipo di evento generico.

[10] cioè l'evento `Expose` non viene gestito

4.3.2 Una prima soluzione

Guardando il sorgente della libreria SDL si può notare che in varie parti all'interno del codice vengono eseguite alcune operazioni soltanto se l'applicazione SDL è eseguita in una propria finestra.

Una prima soluzione potrebbe essere quella di cercare di far eseguire queste operazioni all'applicazione SDL. Attraverso la funzione `SDL_GetWMInfo()` è possibile accedere alle informazioni private relative alla finestra e al `Display` assegnate da SDL. Una volta ottenute queste informazioni, è possibile effettuare una chiamata alla funzione `XSelectInput()` per chiedere al server X di inviare gli eventi relativi alla finestra all'applicazione SDL.

La nuova applicazione si chiama `sdlEv`. É stata creata la funzione `setwinevents()` che viene eseguita quando ci si accorge che l'applicazione deve usare una finestra già esistente. Il codice 4.2 mostra tale funzione più in dettaglio.

Listing 4.2: *Parte della funzione `setwinevents()`*

```
1  static void setwinevents() {
2          [...]
3          XWindowAttributes attr;
4          long event_mask;
5          SDL_SysWMinfo info;
6          SDL_VERSION(&info.version);
7          Display *SDL_Display = info.info.x11.display;
8          Window win = info.info.x11.window;
9
10         event_mask = KeyPressMask | KeyReleaseMask | ButtonPressMask |
11                             ButtonReleaseMask | [...];
12
13         XGetWindowAttributes(SDL_Display, win, &attr);
14         if (not_yours(&attr, ButtonPressMask)) {
15                 fprintf(stderr, "Buttons already taken, removing\n");
16                 event_mask &= ~ButtonPressMask;
17         } [...]
18         event_mask |= attr.your_event_mask;
19         XSelectInput(SDL_Display, win, event_mask);
20 }
```

Esistono alcuni eventi[11] che possono essere gestiti soltanto da un'applicazione. La variabile `event_mask` contiene tutti gli eventi che si desidera siano gentiti da SDL, mentre in `attr` ci sono gli attributi attuali della finestra (e quindi anche gli eventi già catturati dall'applicazione host). La funzione `not_yours()` confronta tali maschere e nel caso in cui l'evento *single recipient* sia già gestito viene eliminato dalla maschere degli eventi da far gestire ad SDL. La chiamata alla funzione `XSelectoInput()` effettua l'assegnamento di tali eventi all'applicazione SDL.

[11] Si veda il manuale della funzione `XSelectInput()`

Esecuzione

Vediamo adesso come si comporta questa applicazione quando eseguita nella finestra del plugin. Il plugin è lo stesso dell'esempio precedente (ovvero `plugin2`), e nell'attributo `progtorun` mettiamo questa volta `sdlEv`. L'applicazione SDL si accorgerà[12] che viene eseguita in una finestra già esistente ed eseguirà la funzione `setwinevents()` quando necessario.

Gli eventi della tastiera sono gestiti correttamente dall'applicazione SDL. Premendo i tasti freccia su/giu è possibile cambiare il colore del rettangolo. Gli eventi del mouse sono gestiti soltanto in parte. Infatti se guardiamo le informazioni di debug mostrate sul terminale ci si accorge che l'evento *pressione del mouse* è già gestito dalla finestra del plugin e quindi non può essere gestito da SDL. Infatti premendo con il mouse all'interno del rettangolo, il *fuoco* passa temporaneamente al plugin. Uscendo dal rettangolo e tornando sopra sarà nuovamente possibile gestire gli eventi della tastiera da parte dell'applicazione SDL.

Per consentire una corretta e completa gestione del mouse è necessario modificare il plugin e rilasciare la gestione degli eventi di pressione del mouse. Questa modifica è stata applicata, insieme ad altre modifiche, nel `plugin3`.

4.3.3 La soluzione finale

Con l'esempio precedente non è ancora possibile gestire il ridimensionamento della finestra. Infatti guardando ancora il sorgente della libreria SDL si nota che la funzione di resize non è effettuata se non si usa una propria finestra. Un altro problema è informare la finestra del plugin quando deve essere ridimensionata.

La prima modifica la faremo al plugin. Per consentire il ridimensionamento metteremo il plugin in attesa della ricezione di un evento di `ConfigureNotify` che viene in genere mandato dalle applicazioni quando è necessario cambiare le geometrie della finestra[13]. Per effettuare il resize del plugin utilizzeremo la capacità di *scripting* del plugin e andremo a modificare le variabili `width` e `height` associate al plugin attraverso chiamate *Javascript*.

Per prima cosa alla prima esecuzione della funzione `NPP_SetWindow()` viene installato un handler per gestire l'evento di *ConfigureNotify*. Alla ricezione di un tale evento, se le dimensioni della finestra richiesta differiscono da quella attuale, viene chiamata la funzione `doResize()`. Questa funzione, attraverso chiamate a *Javascript*, modifica le variabili `width` e `height` appropriatamente. In questo modo il plugin è in grado di ridimensionare la propria finestra.

[12] poichè trova la variabile di ambiente `SDL_WINDOWIS` settata.

[13] dimensione, posizione, ...

Come secondo passo è necessario modificare l'applicazione SDL e renderla in grado di inviare un evento di *ConfigureNotify* ogniqualvolta è necessario cambiare dimensioni. Per far ciò utilizziamo l'applicazione `sdlCom`. Come nell'applicazione `sdlEv` è stata applicata la modifica per consentire la gestione degli eventi di tastiera e mouse. Ogni volta che è necessario ridimensionare la finestra SDL, cioè quando viene chiamata la funzione `SDL_SetVideoMode()`, viene invocata una nuova funzione `resize()`. Questa funzione ha lo scopo di creare un nuovo evento di *ConfigureNotify* con le nuove dimensioni della finestra, e di inviarlo, attraverso la funzione di libreria `X11` `XSendEvent()` al plugin.[14] Inoltre viene effettuato il ridimensionamento della finestra SDL che la libreria non effettua con la chiamata alla funzione `XResizeWindow()`.

Esecuzione

Eseguento il `plugin3` con l'attributo `progtorun` settato con l'applicazione `sdlCom` si può notare come gli eventi di mouse e tastiera siano gestiti correttamente dall'applicazione anche all'interno del plugin. Inoltre premendo i tasti destra/sinistra il ridimensionamento funziona correttamente.

In conclusione, per integrare un'applicazione SDL in un plugin è necessario effettuare delle modifiche all'applicazione SDL stessa.

[14] per la precisione l'evento è mandato alla finestra che è gestita dal plugin

Capitolo 5

Il plugin app-wrapper

app-wrapper è un plugin che consente di inglobare applicazione arbitrarie all'interno di una pagina web. Questa versione gira soltanto su Linux/FreeBSD in quanto nel codice si fa ampio uso di primitive a basso livello dipendente dalla piattarfoma in uso (X11).[1]

Le applicazioni che possono essere inglobate senza eccessivi problemi sono quelle sviluppate usando la libreria `SDL`, debitamente patchate, e varie applicazioni basate sul sistema `X Window`, come *xterm*, *mplayer*, *Xnest*, ecc.

5.1 Il file di configurazione

Il plugin *app-wrapper* supporta l'integrazione di varie applicazioni. Per fare ciò è necessario registrare nel browser un MIME type per ogni applicazione che si vuole supportare. Il file `app-wrapper.conf` (codice 5.1) contiene la configurazione di questi MIME types. Ogni riga del file contiene un MIME type nel formato `type/subtype:suffixes:description:[options:]command_line`, dove

type/subtype è il MIME type;

[1] Una versione per l'ambiente Microsoft Windows è attualmente in sviluppo.

Listing 5.1: *File di configurazione* app-wrapper.conf *di esempio.*

```
#--------- start of examples -------------------
application/x-xterm::xterm into a window:xterm -into %w
application/x-asterisk:asterisk:asterisk into a window:/sbin/asterisk -vd
video/x-flv:flv:mplayer plays flv:stream: safe: mplayer -wid %w -
video/divx:divx:divx mplayer: stream: safe: mplayer -wid %w -
```

Listing 5.2: *Struttura dati per il plugin app-wrapper.*

```
struct _instance {
        int local_href; /* set if href is file:// ... */
        int stream_on_stdin;    /* can handle stream on stdin */
        int pipe_fd[2]; /* in case we open a pipe on a stream... */
        pid_t child_pid;
        long window;    /* the window used by setwindow */
        char cmdbuf[256]; /* a buffer to copy the external command */
        int width, height; /* width and height of plugin window */
};
```

suffixes è una lista di estensioni, separate da una virgola, supportate dal MIME type;

description è la descrizione del MIME type che è mostrata dal browser accedendo alla pagina `about:plugins`;

option è un parametro opzionale (vedi sotto);

command_line è il comando da eseguire, dove a %w è sostituito il window id della finestra del plugin.

Le opzioni disponibili attualmente sono due:

safe: informa il plugin che il MIME type indicato è considerato sicuro e può eseguire qualsiasi URL. Per questioni di sicurezza un MIME type è attivo per default soltanto per URL locali[2].

stream: l'oggetto indicato nell'attributo `src` o `data` è passato allo *stdin* dell'applicazione.

Come detto in precedenza all'avvio del browser viene controllato nelle directory predefinite l'esistenza di nuovi plugin e in caso affermativo viene chiamata la funzione `NP_GetMIMEDescription()`. La modifica del file di configurazione richiede che il browser richiami tale funzione in modo da poter caricare eventuali nuovi MIME types. Per forzare ciò è necessario ad ogni modifica del file di configurazione aggiornare il timestamp del plugin[3] in modo tale che il browser richiami correttamente la funzione `NP_GetMIMEDescription()` all'avvio.

[2] nella forma `file:///`

[3] per esempio con un comando del tipo `'touch app-wrapper.so'`

Listing 5.3: *La funzione `NP_GetMIMEDescription()` per il plugin app-wrapper.*

```
1 char *NP_GetMIMEDescription(void) {
2         static char mime_buf[2048];
3         char *s;
4         bzero(mime_buf, sizeof(mime_buf));
5         s = check_config(NULL, NULL, mime_buf, sizeof(mime_buf));
6         if (s == NULL)
7                 s = "";
8         return s;
9 }
```

5.2 Struttura dati

La struttura dati per il plugin *app-wrapper* è mostrata nel codice 5.2.

local_href è un *boolean* che indica se l'url è locale o remota. È usato per testare se stiamo eseguendo un comando *sicuro* o no;

stream_on_stdin è un *boolean* che indica se è necessario inviare il file specificato nell'attributo `src` o `data` allo *stdin*;

window è un intero che contiene il *window id* della finestra del plugin dove verrà eseguita l'applicazione esterna.

Gli altri campi sono autoesplicativi e non necessitano ulteriore aprofindimento.

5.3 A grandi linee

Nella figura 5.1 è mostrato uno schema con il funzionamento a grandi linee del plugin `app-wrapper`. In linea di principio il flusso del plugin è simile a quello degli esempi mostrati in precedenza, con la sostanziale differenza che molte funzioni delle `NPAPI` hanno degli scopi ben precisi.

- La funzione `NP_GetMIMEDescription()` ha lo scopo di interpretare il file di configurazione alla ricerca di tutti i MIME types gestiti dal plugin.
- L'inizializzazione globale non effettua niente di particolare, mentre nella funzione `NPP_New()` si effettuano importanti operazioni, quali controllare se stiamo eseguendo un url locale o remota, individuare il MIME type associato all'istanza del plugin e scorrere il file di configurazione alla ricerca del comando per eseguire l'applicazione associata.

- La funzione `NPP_SetWindow()` prepara la finestra associata al plugin ad eseguire l'applicazione. Vengono deselezionati gli eventi gestiti dal plugin per default cosicchè possano essere gestiti dall'applicazione esterna, viene installato l'handler per la gestione del ridimensionamento della finestra, e viene poi chiamata la funzione `run_cmd()` che avvierà l'applicazione.
- Dopo l'avvio dell'applicazione, il plugin si mantiene in attesa di eventuali eventi di resize, invocando quando necessario la funzione `doResize()`.
- La distruzione dell'istanza è effettuata dalla funzione `NPP_Destroy()`, che controlla se l'applicazione è ancora in esecuzione e in caso affermativo gli viene inviato un signale di terminazione.[4]

5.4 Compilazione e Installazione

Per la compilazione del plugin sono necessari, oltre ai file forniti nel pacchetto[5], i file di sviluppo di X11 e SDL. Tipicamente questi file sono rilevati correttamente dal `Makefile` fornito, in caso di necessità è sufficiente modificare le linee `-I...` e `-L...` con i percorsi di suddetti files.

Le macro utilizzate sono le seguenti:

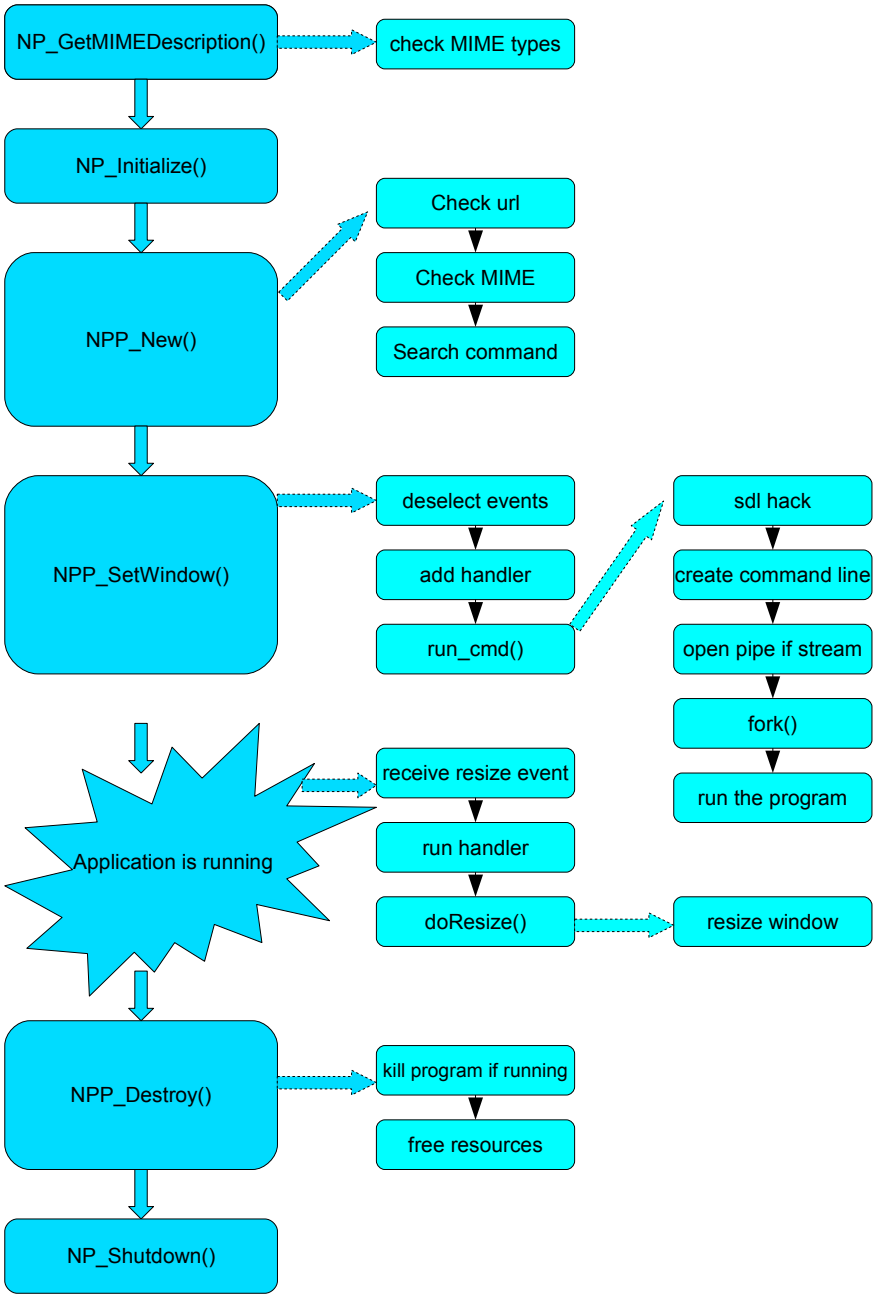
- `PLUGINDEBUG`
- `USEINITIAL`
- `WANT_X11`
- `WANT_SDL`
- `WANT_STREAM`

Una volta dato il comando `Make`, verrà creato il file `app-wrapper.so` che è la libreria contenente il plugin, da copiare, insieme al file di configurazione `app-wrapper.conf` nella directory dei plugin.

[4] viene inviato un segnale di SIGTERM per consentire una chiusura regolare dell'applicazione.

[5] come specificato nel capitolo 2

Figura 5.1: *Schema del funzionamento del plugin* `app-wrapper`*. Sulla sinistra sono mostrate le principali funzioni facenti parti di* `NPAPI`*, mentre sulla destra sono specificate le ulteriori funzioni chiamate, o le operazioni che tale funzione esegue.*



Listing 5.4: *La funzione `NPP_New()` per il plugin app-wrapper.*

```
NPError NPP_New(NPMIMEType pluginType, NPP instance,
                uint16 mode, int16 argc, char *argn[],
                char *argv[], NPSavedData *saved) {
        [...]
        if (!plugin_calls_js(instance, "document.location.href", &res))
                fprintf(stdedd, "Error on javascript call\n");
        else {
                fprintf(stderr, "f returns [%s]\n",
                        NPVARIANT_TO_STRING(res).utf8characters);
                if (NPVARIANT_IS_STRING(res)) {
                        const char *x = NPVARIANT_TO_STRING(res).utf8characters;
                        if (!strncasecmp(x, "file://", 7))
                                my_instance->local_href = 1;
                }
                NPN_ReleaseVariantValue(&res);
        }
        for (i = 0; i < argc; i++) {
                int l = sizeof(my_instance->cmdbuf) - 1;
                char *s;
                fprintf(stderr, "Argument %d %s = '%s'\n",
                        i, argn[i], argv[i]);
                /* store the translated type in a local string */
                if (!strcmp(argn[i], "type") &&
                        (s = check_config(argv[i], my_instance,
                                my_instance->cmdbuf, l)) ) {
                        strncpy(my_instance->cmdbuf, s, l);
                        my_instance->cmdbuf[l] = '\0';
                }
        }
        return NPERR_NO_ERROR;
}
```

5.5 Inizializzazione

5.5.1 La funzione `NP_GetMIMEDescription()`

La funzione `NP_GetMIMEDescription()` (codice 5.3) si occupa di caricare i vari MIME types associati al plugin. Come detto in precedenza è necessario forzare il browser a eseguire questa funzione ogni volta che vengono aggiunti o eliminati MIME types dal file di configurazione. Questa funzione chiama un'ulteriore funzione (`check_config()`) che si occupa di interpretare il file di configurazione alla ricerca di MIME types. In questo caso viene passato un puntatore nullo all'istanza, in modo tale da richiedere l'elenco dei MIME types supportati. Si veda l'appendice A.1 per maggiori informazioni sulla funzione `check_config()`.

Listing 5.5: *La funzione `NPP_SetWindow()` per il plugin app-wrapper.*

```
NPError NPP_SetWindow(NPP instance, NPWindow* window) {
    struct _instance *me;
    NPSetWindowCallbackStruct *ws_info;
    XWindowAttributes attr;
    me = instance->pdata;
    ws_info = window->ws_info;
    if (me->window == 0) {
        long private_events = ButtonPressMask | SubstructureRedirectMask;
        me->window = (long)window->window;
        bzero(&attr, sizeof(attr));
        XGetWindowAttributes(ws_info->display, me->window, &attr);

        if (attr.your_event_mask & private_events) {
                attr.your_event_mask &= ~private_events;
                XSelectInput(ws_info->display, me->window,
                        attr.your_event_mask);
        }

        Widget xtwidget = XtWindowToWidget(ws_info->display, me->window);
        if (xtwidget) {
                XtAddEventHandler(xtwidget, StructureNotifyMask, False,
                        (XtEventHandler)xt_event_handler, instance);
        }
        run_cmd(me);
    }
    [...]
    return NPERR_NO_ERROR;
}
```

5.5.2 Creazione dell'istanza

La funzione `NPP_New()` (codice 5.4) dopo l'allocazione e l'inizializzazione della memoria per l'istanza, controlla se l'url è locale o meno attraverso una chiamata `document.location.href` di *Javascript*. Dopo viene chiamata la funzione `check_config()` (vedi A.1) passandogli il MIME type in modo che la questa funzione ritorni il comando da eseguire per avviare l'applicazione corrispondente al MIME types passatogli.

5.5.3 Esecuzione del plugin

Finita l'esecuzione della funzione `NPP_New()` viene invocata la funzione `NPP_SetWindow()`.[6]

L'applicazione da integrare nel plugin viene eseguita la prima volta che il browser chiama la funzione `NPP_SetWindow()` (codice 5.5). Prima di eseguire il comando, è necessario preparare la finestra del plugin per l'integrazione dell'applica-

[6] Dopo eventuali chiamate a `NPP_GetValue()`

Listing 5.6: *La funzione* `doResize()` *per il plugin app-wrapper.*

```
bool doResize(NPP instance, int w, int h) {
        NPObject *idObj = NULL; /* the id object */
        NPVariant width, height; /* New width and height to pass to browser */
        err = NPN_GetValue(instance, NPNVPluginElementNPObject, &idObj);
        INT32_TO_NPVARIANT(w, width);
        INT32_TO_NPVARIANT(h, height);
        retval = NPN_SetProperty(instance, idObj,
                        NPN_GetStringIdentifier("width"), &width);
        retval = NPN_SetProperty(instance, idObj,
                        NPN_GetStringIdentifier("height"), &height);
        if (idObj)
                NPN_ReleaseObject(idObj);
        return retval;
}
```

zione esterna. La variabile `private_events` contiene le maschere degli eventi che possono essere gestiti soltanto da un'applicazione alla volta[7]. In `attr` si memorizzano tutti gli attributi della finestra del plugin attraverso la chiamata alla funzione `XGetWindowAttributes()`, e si controlla se gli attributi *privati* sono già settati per questa finestra. In caso affermativo si deselezionano (con la chiamata alla funzione `XSelectInput()`) in modo tale che questi eventi possano essere gestiti dall'applicazione esterna.

Successivamente è creato un *handler* per l'evento *ConfigureNotify*. Questo evento è generato dalle applicazioni quando necessitano di modificare le dimensioni della propria finestra. Alla ricezione di un tale evento viene eseguita la funzione `xt_event_handler()` che effettua il ridimensionamento della finestra del plugin.

A questo punto la finestra del plugin è pronta per ospitare l'applicazione; la funzione `run_cmd()` controlla se si hanno permessi per poter eseguire l'applicazione, crea i vettori con le opzioni per l'applicazione e la esegue.

Per quanto riguarda le applicazioni basate sulla libreria SDL, è necessario modificare il codice per renderla in grado di generare l'evento di *ConfigureNotify* ogni qualvolta è necessario ridimensionare la propria finestra (si veda il paragrafo 4.3.3).

5.5.4 Il ridimensionamento della finestra

Il ridimensionamento della finestra del plugin è effettuato alla ricezione dell' evento `ConfigureNotify` che solitamente viene inviato da un'applicazione ogni volta che necessita di modificare la geometria della propria finestra. L'*handler* installato in pre-

[7] Si veda il manuale unix della funzione `XSelectInput()`

cedenza[8] si occupa di catturare questo evento e di chiamare la funzione `doResize()` (vedi il codice 5.6) che si occupa del ridimensionamento della finestra.

Il ridimensionamento è effettuando agendo sulle variabili `width` e `height` assegnate al plugin dal DOM. Con la chiamata di funzione `NPN_GetValue(instance, NPNVPluginElementNPObject, &idObj)` si recupera l'identificatore del plugin e, dopo aver convertito le variabili da intero a `NPVariant`, si settano attraverso la chiamata a `NPN_SetProperty`. Infine si rilascia l'oggetto `idObj` attraverso la funzione `NPN_ReleaseObject`.

Come detto in precedenza è necessario modificare le applicazioni SDL per renderle in grado di generare un evento di *ConfigureNotify* ogniqualvolta è necessario ridimensionare la finestra.

[8] nella funzione `NPP_SetWindow()`

Capitolo 6

La versione per Microsoft Windows

Vediamo in questo capitolo le modifiche che si devono effettuare per scrivere un plugin compatibile con Microsoft Windows.

Inizialmente vedremo le differenze sull'uso delle NPAPI, poi analizzeremo il caso delle applicazioni basate sulla libreria SDL e infine mostreremo un prototipo del plugin app-wrapper per Microsoft Windows.

6.1 NPAPI: le differenze con Linux/FreeBSD

Sebbene le NPAPI siano multipiattaforma, un plugin scritto per Linux/FreeBSD potrebbe necessitare di modifiche per permettere la compilazione su Windows.

In particolare, oltre alla mancanza di alcuni costrutti tipici di Unix, come ad esempio la fork(), è presente una lieve differenze nelle NPAPI

6.1.1 Inizializzazione

L'inizializzazione di un plugin è differente in ambiente Microsoft Windows. Non è presente la funzione NP_GetMIMEDescription() e la funzione NP_Initialize() ha un prototipo diverso.

La funzione NP_GetEntryPoints()

In Windows la prima funzione invocata nel momento del caricamento del plugin è NP_GetEntryPoints. Questa funzione invia al browser la tabella delle funzioni NPP_* implementate nel plugin[1] operazione che in ambiente Unix era effettuata dalla funzione NP_Initialize().

[1] NPPluginFuncs*

Se si usano i files forniti in questo progetto, in questa funzione, essendo in assoluto la prima funzione chiamata dal browser, viene creato l'eventuale file per il debug del plugin.

La funzione NP_Initialize()

Poiche come detto in precedenza la tabella delle funzioni NPP_* è riempita nella funzione NP_GetEntryPoints(), la funzione NP_Initialize() effettua soltanto il salvataggo della tabella delle funzioni NPN_* esportate dal browser.

Seguendo le linee guida introdotte nel capitolo 2, lo sviluppatore non deve tener conto di queste differenze in quanto tutte queste operazioni di inizializzazione sono effettuate automaticamente. Nel caso fosse necessario utilizzare funzioni di inizializzazione e chiusura personalizzate, i puntatori alle due funzioni hanno lo stesso prototipo qualsiasi sia il sistema operativo.

I MIME types

Come detto in precedenza in Microsoft Windows il browser non chiama la funzione NP_GetMIMEDescription(). I MIME types supportati dal plugin, con le estensioni associate, devono essere specificati nelle *proprietà* della dll.[2] Si veda il file windows/npbase.c per ulteriorei informazioni.

6.2 Applicazioni SDL: embedding in una finestra esistente

Anche in ambiente Windows, SDL mette a disposizione la possibilità di inglobare un'applicazione in una finestra esistente attraverso l'uso della variabile d'ambiente SDL_WINDOWID. Anche in questo caso, l'applicazione non sarà in grado di gestire alcun evento.

In ambiente Microsoft Windows ho usato un approccio diverso per inglobare un'applicazione SDL in una finestra già esistente (un plugin). Invece di usare la variabile di ambiente, l'applicazione SDL viene avviata normalmente creando una propria finestra. A questo punto alla finestra viene assegnato un nuovo *padre* attraverso la chiamata alla funzione SetParent() facente parte delle WINAPI. In questo modo la finestra SDL viene inglobata nella finestra già esistente. Effettuando l'embedding in questa maniera, tutti gli eventi sono gestiti regolarmente da SDL in quanto dal punto di vista della libreria è come se l'applicazione fosse eseguita in una propria finestra.

[2] In particolare devono essere specificati almeno i seguenti campi: MIMEType, FileDescription, FileExtents.

Listing 6.1: *Parte di codice dell'applicazione* `sdlwin.exe`.

```
[...]
  HWND SDL_Window = info.window;
  if (argc > 1) {
    DWORD style;
    /* Take the sdl window attributes */
    style = GetWindowLong(SDL_Window, GWL_STYLE);
    /* and remove the unneed attributes to embed this window in the window */
    style &= ~(WS_CAPTION|WS_OVERLAPPEDWINDOW |WS_SYSMENU); /*Remove: title, */
    style &= ~(WS_POPUP | WS_BORDER);
    /* Add some mandatory window attributes*/
    style |= WS_CHILD;
    style |= WS_MAXIMIZE;
    /* Set new attributes */
    SetWindowLong(SDL_Window, GWL_STYLE, style);
    /* We must call SetWindowPos() to update the window */
    SetWindowPos(SDL_Window, NULL, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE
                                    | SWP_NOZORDER | SWP_FRAMECHANGED);

    /* The same as above, in the existing window */
    style = GetWindowLong(win, GWL_STYLE);
    style |= (WS_OVERLAPPEDWINDOW|WS_CAPTION|WS_SYSMENU|WS_MINIMIZEBOX);
    style |= (WS_VSCROLL | WS_HSCROLL);
    style |= WS_VSCROLL;
    SetWindowLong(win, GWL_STYLE, style);

    /* Embed the sdl window into existing window */
    SetParent(SDL_Window, win);
  }
  [...]
```

Ci sono alcuni problemi ad effettuare questa operazione: in particolare è necessario eliminare tutte le decorazioni della finestra sdl in quanto altrimenti comparirebbero all'interno della nuova finestra. É necessario tener conto dell'eventuale presenza di *scroll bars* per determinare le corrette dimensioni della finestra padre.

6.2.1 L'applicazione SDL

Il funzionamento dell'applicazione SDL di esempio è lo stesso di quella usata per gli esempi su Linux/FreeBSD. Per effettuare l'embedding non si usa la variabile d'ambiente `SDL_WINDOWID` ma l'id della finestra viene passato come argomento dell'applicazione. Se la variabile d'ambiente è settata, SDL effettuerà il proprio embedding e non sarà possibile gestire alcun evento.

Come si nota dal codice 6.1, dopo l'inizializzazione di SDL (omessa nel codice qui riportato), se è passato un *windowid* come argomento dell'applicazione[3] vengono

[3] cioè se argc è maggiore di 1 supponiamo che il primo argomento sia l'identificatore della finestra, non vengono effettuati controlli sulla correttezza di questo valore.

effettuate alcune operazioni. Si recupera lo stile della finestra di SDL e dopo aver rimosso gli attributi non necessari (eventuali menù, i bordi, le icone, ...), si massimizza la finestra. Gli attributi vengono settati con la funzione `SetWindowLong()`, e con la funzione `SetWindowPos()` vengono resi attivi. Successivamente viene modificato lo stile della finestra host, aggiungendo anche le barre per lo scrolling, oltre al menù e alle icone di minimizzazione.[4] Infine, con la chiamata a `SetParent()` la finestra di SDL viene inglobata all'interno di quella esistente.

Per quanto riguarda il ridimensionamento, la finestra di SDL viene ridimensionata dall'applicazione SDL, mentre viene mandato un messaggio (`WM_SIZE`) alla finestra host (il plugin) per ridimensionare la finestra padre. Le dimensioni delle due finestre possono non essere le stesse, infatti è necessario tener conto di eventuali decorazioni presenti.

Il plugin utilizzato per l'embedding è `npbase.dll`, descritto nel paragrafo successivo.

6.3 Il plugin npbase.dll

A grandi linee questo plugin è simile alla versione per Linux. Il ridimensionamento della finestra viene eseguita modificando le variabili `width` e `height` attraverso lo *scripting*, mentre le dimensioni della nuova finestra vengono ottenute attraverso le ricezione di un messaggio inviato dall'applicazione SDL.

Vedremo nei paragrafi successivi le differenze più importanti fra le due versioni.

6.3.1 La funzione `NPP_SetWindow()`

Parte della funzione `NPP_SetWindow()` è mostrata nel codice 6.2.

Listing 6.2: *La funzione `NPP_SetWindow()` per il plugin `npbase`.*

```
1   [...]
2   me->window = (HWND)window->window;
3   /* Subclass window so we can intercept window messages */
4   lpOldProc = SubclassWindow(me->window, (WNDPROC)PluginWinProc);
5   RECT bounds = {0, 0, 0, 0,};
6   int w = me->width;
7   int h = me->height;
8   bounds.right = w;
9   bounds.bottom = h;
10  LONG style = GetWindowLong(me->window, GWL_STYLE);
11  AdjustWindowRectEx(&bounds, style, (GetMenu(me->window) != NULL), 0);
12  w = bounds.right-bounds.left;
```

[4] Questi stili sono settati nella finestra padre in quanto così facendo è possibile inglobare questa applicazioni in una finestra esistente vuota, creando anche i menù e tutti gli ornamenti normalmente presenti. Per inglobare tale finestra all'interno del plugin, questi stili verrano eliminati.

```
13      h = bounds.bottom-bounds.top;
14      SetWindowLongPtr(me->window, GWL_STYLE, style);
15      SetWindowPos(me->window, NULL, 0, 0, w, h, SWP_NOMOVE | SWP_NOSIZE |
        SWP_NOZORDER | SWP_FRAMECHANGED );
16      /* Associate window with our instance so we can access it in the window
17       * procedure */
18      SetWindowLongPtr(me->window, GWL_USERDATA, (LONG)instance);
```

La prima volta che questa funzione è chiamata, viene effettuato il *subclassing* della finestra (attraverso la funzione `SubclassWindow()`) in modo tale da poter gestire gli eventi della finestra. La funzione `PluginWinProc()` agirà come *handler* per l'evento di resize ricevuto dall'applicazione SDL. Dopo sono estratti gli stili della finestra in modo da calcolare le dimensioni corrette della finestra tenendo conto degli stili applicati (`AdjustWindowRectEx()`). Infine con la funzione `SetWindowLongPtr()` si memorizza il puntatore all'istanza del plugin nello spazio riservato alla finestra in modo tale da potervi accedere sempre.

6.3.2 L'handler per l'evento `WM_SIZE`

Il subclassing della finestra del plugin ci permette di poter catturare gli eventi inviati alla finestra. A noi interessa l'evento `WM_SIZE`, che viene trattato nella funzione `PluginWinProc()` (codice 6.3).

Listing 6.3: *La funzione che agisce come handler per l'evento `WM_SIZE`*

```
1   [...]
2       RECT bounds = {0, 0, 0, 0,};
3       bounds.left = 0;
4       bounds.top = 0;
5       bounds.right = LOWORD(lParam);
6       bounds.bottom = HIWORD(lParam);
7       LONG style = GetWindowLong(hWnd, GWL_STYLE);
8       /* Remove all window decorations */
9       style &= ~(WS_OVERLAPPEDWINDOW|WS_CAPTION|WS_SYSMENU|WS_MINIMIZEBOX|WS_POPUP
        );
10      style |= (WS_VSCROLL | WS_HSCROLL | WS_CHILD);
11      /* calculate the right size for the window */
12      AdjustWindowRectEx(&bounds, style, (GetMenu(hWnd) != NULL), 0);
13      int w = bounds.right-bounds.left + GetSystemMetrics(SM_CYHSCROLL);
14      int h = bounds.bottom-bounds.top + GetSystemMetrics(SM_CXVSCROLL);
15      /* Set new window style */
16      SetWindowLong(hWnd, GWL_STYLE, style);
17      doResize(instance, w, h);
18      return 0;
19    }
20  [...]
```

Dopo aver estratto le nuove dimensioni della finestra, ne eliminate tutte le decorazioni della finestra, e aggiunte le *scroll bars*[5]. Una volta settato il nuovo stile, viene chiamata la funzione `doResize()` che effettua il ridimensionamento della finestra del plugin, come avviene nella versione per Linux/FreeBSD (si veda il paragrafo 5.5.4[6])

[5] non c'è un motivo particolare per averle aggiunte, è soltnto un modo per mostrare che è necessario tener conto dell'eventuale presenza delle *scroll bar* per calcolare le giuste dimensioni della finestra.

[6] con le ovvie differenze tra i nomi di messaggi ricevuti.

Appendice A

Alcune funzioni in dettaglio

Vediamo adesso con un maggiore dettaglio alcune delle funzioni chiave riguardanti il plugin `app-wrapper`.

A.1 La funzione `check_config()`

La funzione `check_config()` è chiamata all'interno del plugin con due scopi differenti:

- dalla funzione `NP_GetMIMEDescription()` per estrarre dal file di configurazione l'elenco del MIME types supportato dal plugin;
- dalla funzione `NPP_New()` per controllare le opzioni del MIME type passato dal browser all'istanza del plugin e per estrarre il comando da eseguire associato.

A.1.1 Estrazione dei MIME types

Come detto in precedenza la funzione `NP_GetMIMEDescription()` ha lo scopo di estrarre dal file di configurazione `app-wrapper.conf` l'elenco dei MIME types supportati dal plugin. Per fare il parsing del file di configurazione è utilizzata la funzione `check_config()`, chiamata con i parametri `mime` e `_instance` uguali a `NULL`.

Listing A.1: *La ricerca dei MIME type supportati dal plugin appwrapper.*

```
1 static char * check_config(char *mime, struct _instance *me,
2                  char *buf, int bufsize) {
3         FILE *f = NULL;
4         struct passwd *pwe = NULL, pwe_buf;
5         char *fname = NULL;
6         char *ret = NULL;
7         int i, l;
```

```
8           uid_t my_uid;
9           char *bufp;
10
11          if (mime == NULL)
12                  mime = "GET_MIME";
13
14          /* retrieve uid, home directory, and try to open the config file. */
15          my_uid = getuid();
16          bzero(&pwe_buf, sizeof(pwe_buf));
17          if (my_uid <= 0)
18                  goto done;
19          i = getpwuid_r(my_uid, &pwe_buf, buf, bufsize, &pwe);
20          endpwent();
21          if (i != 0 || pwe == NULL || pwe->pw_dir == NULL)
22                  goto done;
23          asprintf(&fname, "%s/.mozilla/plugins/app-wrapper.conf", pwe->pw_dir);
24          if (fname == NULL)
25                  goto done;
26          f = fopen(fname, "r");
27          if (f == NULL)
28                  goto done;
29
30          /* read from the file and look for a match in the mime type. */
31          l = strlen(mime);
32          buf[0] = '\0';  /* terminate the response */
33          bufp = buf;     /* append the result here */
34          for (;;) {
35                  char *d, *cmd, *end = NULL, line[1024];
36                  int is_safe = 0;
37                  d = fgets(line, sizeof(line), f);
38                  if (d == NULL)
39                          break;
40                  /* skip leading whitespace */
41                  d = skip_blanks(d);
42                  if (index("#\r\n", *d))
43                          continue;       /* empty line */
44                  /* The third ':' is the end of the mime type. */
45                  cmd = index(d, ':');
46                  if (cmd)
47                          cmd = index(cmd + 1, ':');
48                  if (cmd)
49                          cmd = index(cmd + 1, ':');
50                  if (cmd) {
51                          for (end = cmd; *end && !index("\r\n#", *end); end++)
52                                  ;
53                          *end = '\0';
54                  }
55                  if (me == NULL) { /* GetMIMETypes */
56                          /* copy the mime line in the buffer,
57                           * replace ':' with ',' after cmd
58                           */
59                          for (; cmd && *cmd; cmd++) {
60                                  if (*cmd == ':')
61                                          *cmd = ',';
62                          }
63                          if (bufsize < 4) {
64                                  fprintf(stderr, "no space, skip [%s]\n", d);
65                                  continue;
66                          }
67                          l = strlen(d);
```

```
68                          if (l > bufsize - 2)
69                                  l = bufsize - 2;
70                          strncat(bufp, d, l);
71                          bufp[l++] = ';';
72                          bufp[l] = '\0';
73                          bufp += l;
74                          bufsize -= l;
75                          continue;
76                  }
77                  [...]
78          }
79  done:
80          if (me == NULL && buf[0] != '\0')
81                  ret = buf;
82          if (ret == NULL)
83                  fprintf(stderr, "Error checking config for %s uid %d\n",
84                          mime, my_uid);
85          else
86                  fprintf(stdout, "[%s] maps to [%s]\n", mime, ret);
87          if (f)
88                  fclose(f);
89          if (fname)
90                  free(fname);
91          return ret;
92  }
```

Il codice A.1 mostra la parte della funzione incaricata di estrarre i MIME type supportati dal file di configurazione.

All'inizio della funzione viene cercato il percorso del file di configurazione. Dalla linea 15 viene cercato l'utente che sta eseguendo l'istanza del plugin, estratto il percorso della *home directory*e successivamente viene aperto in lettura il file di configurazione (linea 26). A questo punto inizia il ciclo che effettua il parsing del file di configurazione. Viene letta una riga alla volta del file di configurazione[1]. Se la riga è vuota o inizia per il carattere #[2] (linea 42) viene ignorata e si passa alla riga successiva. A questo punto ci portiamo alla fine del MIME type (dopo il terzo `':'` della riga, linea 49) e rimpiazziamo gli eventuali due punti che incontriamo da ora in poi con una virgola[3]. Poi concateniamo la riga così modificata nel buffer passato nella chiamata della funzione (linea 70) e aggiungiamo un punto e virgola per eventuali MIME types aggiuntivi. Se non sono occorsi (linea 81) ritorniamo l'intero elenco di MIME type alla funzione `NP_GetMIMEDescription()`.

[1] La lunghezza massima della riga accettata è di 1024 caratteri, valore considerato adeguato.

[2] ricordiamo che il carattere # è il delimitatore per i commenti.

[3] questo perchè i due punti sono un delimitatore per i MIME types gestiti da Mozilla, ma i due punti a questo punto della riga indicano le eventuali opzioni. In questo caso le opzioni compariranno nella descrizione del MIME type.

A.1.2 Estrazione delle opzioni e del comando da eseguire

Nella funzione `NPP_New()` avviene la seconda chiamata alla funzione `check_config()`. In questo caso viene passato un MIME type, il puntatore all'istanza e un buffer. Il valore di ritorno è il comando da eseguire per avviare l'applicazione appartentente al MIME type passato. Nel codice A.2 è mostrata la parte della funzione che effettua la ricerca del MIME type dal file di configurazione, controlla le varie opzioni e ritorna il comando da eseguire.

Listing A.2: *Estrazione delle opzioni e del comando da eseguire la file di configurazione* `appwrapper.conf`.

```
1  [...] parte precedente fino a linea 54 del codice A.2
2                  if (strncasecmp(d, mime, l) || d[l] != ':')
3                          continue; /* mime type mismatch */
4                  if (!cmd)
5                          continue;
6                  cmd = skip_blanks(cmd+1);       /* point to the command */
7                  for (;;) {
8                          if (!strncasecmp(cmd, "safe:", 5)) {
9                                  is_safe = 1;
10                                 cmd = skip_blanks(cmd + 5);
11                         } else if (!strncasecmp(cmd, "stream:", 7)) {
12                                 fprintf(stdout, "--- can do streaming\n");
13                                 me->stream_on_stdin = 1;
14                                 cmd = skip_blanks(cmd + 7);
15                         } else {
16                                 break;
17                         }
18                 }
19                 /* skip empty lines */
20                 if (*cmd == '\0' || index("#\r\n", *cmd)) {
21                         ret = NULL;
22                 } else if (!is_safe && !me->local_href) {
23                         fprintf(stdout, "unsafe command %s\n", cmd);
24                         ret = NULL;
25                 } else {
26                         strncpy(buf, cmd, bufsize - 1);
27                         buf[bufsize-1] = '\0';
28                         ret = buf;
29                 }
30                 break;
31         }
32 [...] continua in codice A.1 linea 79
```

Alla linea 2 viene controllata se la riga corrente contiene il MIME type passato alla funzione e in caso negativo si ritorna all'inizio del ciclo (linea 34 del codice A.1) per esaminare la riga successiva. Se il MIME type corrisponde, esaminiamo il proseguo della linea per controllare se sono specificate delle opzioni, ed eventualmente si settano le rispettive variabili dell'istanza (linea 8 e seguenti). Alla linea 22 ci si assicura di non eseguire un'applicazioneche non sia stata marcata come *safe* in caso di URL

Listing A.3: *Esecuzione di codice Javascript.*

```
1  static bool plugin_calls_js(NPP instance, const char *js_code, NPVariant *res)
2  {
3          NPError err;
4          bool retval = FALSE;
5          NPObject *winObj = NULL; /* the browser window */
6          NPString str;   /* NS representation of the javascript code */
7          err = NPN_GetValue(instance, NPNVWindowNPObject, &winObj);
8          if (err != NPERR_NO_ERROR) {
9                  fprintf(stderr, "%s GetValue NPNVWindowNPObject fails\n",
10                         __FUNCTION__);
11                 goto done;
12         }
13         str.utf8characters = js_code;
14         str.utf8length = strlen(str.utf8characters);
15         bzero(res, sizeof(*res));
16         retval = NPN_Evaluate(instance, winObj, &str, res);
17 done:
18         if (winObj)
19                 NPN_ReleaseObject(winObj);
20         return retval;
21 }
```

non locale. Se non ci sono stati errori (linea 26) si copia il comando[4] per eseguire l'applicazione nel buffer che poi sarà ritornato.

A.2 La funzione `plugin_calls_js()`

La funzione `plugin_calls_js()` (codice A.3) è utilizzata pe eseguire del codice *Javascript* dall'interno del plugin. Il codice da eseguire è passato nel parametro `js_code`, mentre il puntatore `res` contiene l'eventuale risultato dell'operazione, espresso tramite un `NPVariant` che poi dovrà essere convertito al tipo voluto attraverso le macro fornite nelle `NPAPI`.

Per prima cosa si recupera l'identificatore del plugin attraverso la funzione `NPN_GetValue(...,NPNVWindowNPObject,...)` (codice A.3 linea 7), poi creiamo l'istruzione che vogliamo eseguire come `NPString`[5] e attraverso la funzione `NPN_Evaluate()` (linea 16) la eseguiamo. L'eventuale risultato dei questa operazione è memorizzata nel puntatore `res` che sarà poi passato al chiamante.

É importante ricordare che nel codice del chiamante è necessario liberare le risorse occupate dalla variabile di ritorno `res` attraverso una chiamata a `NPN_ReleaseVariantValue()`.

[4] Il puntatore `cmd` punta all'inizio della stringa contenente il nome dell'applicazione da eseguire.

[5] che altro non è che una struttura che contiene una stringa e la sua dimensione

Appendice B

Bibliografia

Oltre alle note e ai riferimenti presenti nel testo sono state consultate le seguenti fonti:

- The Xlib manual:
  http://tronche.com/gui/x/xlib/
- Preparazione ambiente per compilazione applicazioni SDL:
  http://gpwiki.org/index.php/SDL:Tutorials:Setup
- Applicazione SDL di esempio:
  http://www.siforge.org/articles/2004/03/22-sdl-intro-2.html
- Compilazione sotto `Microsoft Windows`:
  https://developer.mozilla.org/en/Compiling_The_npruntime_Sample_Plugin_in_Visual_Studio
- Windows Api Reference:
  http://msdn.microsoft.com/en-us/library/aa383749(VS.85).aspx