Tesina di
Architettura dei sistemi distribuiti
Docente: Antonio Lioy

Bonobo: l'interfaccia Corba di Gnome

Bovero Matteo Onofrio
Matricola: 93152

Typeset by LaTeX

matteo@assi.polito.it
http://www.voyager.it

Indice

Elenco delle tabelle

Elenco delle figure

1 Componentware

Il concetto di riutilizzabilità del software, da sempre fa parte della mentalità del programmatore. Negli ultimi anni, questo concetto si è evoluto fino a concretizzarsi nel neologismo *"componentware"*.

L'idea di base è sempre la stessa, cioè il riutilizzo del codice già scritto, ma portata ad un punto tale che l'utilizzatore finale, il programmatore che scrive applicazioni basate sui componenti, si affida completamente a *"scatole chiuse"* di cui non sa nulla (o quasi), se non il modo in cui si possono interconnettere tra di loro, quali parametri passargli e quali gli vengono restituiti.

Il vantaggio di tutto ciò è palese, sia al programmatore sia al commerciale, infatti grazie a questa tecnologia è possibile accellerare notevolmente la stesura di codice complesso garantendone la funzionalità e l'interoperabilità con applicazioni sviluppate con la stesso tipo di tecnologia.

Il mercato non è rimasto insenbile ai vantaggi tecnici ed economici che il componentware può dare. Infatti siamo oggi difronte ad una vera e propria guerra degli standard. Colui che ne uscirà vincente sarà in grado di imporre i propri modelli di componenti, le rispettive strategie di trattamento e il proprio modello di mercato. [3, B]

1.1 Corba, DCOM o Java/Rmi?

I principali standard attualmente disponibili sul mercato sono Corba, DCOM, e RMI.

1. Corba è uno standard aperto definito dall'OMG e permette l'iteroperabilità tra oggetti remoti tramite un bus software (ORB Object Request Broker). Supporta svariati linguaggi, tutti quelli per cui è previsto un binding, come ad esempio C/C++, Java, Perl, Php, Phyton, Ruby (tramite libpanel-applet-ruby), etc... Il protocollo di comunicazione è IIOP (Internet Inter-Orb Protocol). [4, W]

2. DCOM è il modello di componente proposto da Microsoft. Un'inrfaccia COM definisce il formato in memoria dell'oggetto server che può essere sviluppato nei linguaggi per i quali esiste l'apposito mapping, ciò C++, Visual Basic e C#. É la base per la piattaforma .NET. [3, W]

3. RMI è il modello di programmazione distribuita di Java. Permette lo sviluppo di applicazioni mulpiattaforma ma è limitato al linguaggio della Sun. [1, W]

(Per una comparazione più accurata si rimanda all'Appendice A)

2 Il Pinguino, la Scimmia ed il Drago

In ambiente Linux [2, W] ci sono principalmente due suit di applicativi che permettono una programmazione basata sui componenti. Questi ambienti sono Gnome (basato su Gtk, la Scimmia) e Kde (basato su Qt, il Drago). Un esempio della loro completa integrazione si può avere rispettivamente con Nautilus e Konqueror:

> *"Nautilus e Konqueror, oltre ad eseguire i compiti di un File Manager classico [...] vanno molto oltre: sono un po' dei visualizzatori universali. Ad esempio, selezionate un file e vedrete il contenuto del file visualizzato, sia esso un'immagine, un documento di testo o un filmato; se tutto questo non vi basta, immaginate che funziona anche in remoto: potete quindi chiedere di visualizzare file residenti su Internet [...] Nautilus e Konqueror non fanno assolutamente nulla di tutto quel che vi ho detto: loro si limitano ad essere dei contenitori di componenti di svariata natura. Chiedete ai due programmi di visualizzare una cartella? Loro cercheranno il componente adatto, gli presteranno dello spazio al loro interno e gli chieranno di visualizzare il contenuto della cartella; [...] Agli occhi degli utenti sembra che siano i due Desktop Enviroment a fare tutto il lavoro, in realtà sono i componenti da loro richiamati che si occupano delle operazioni effettive."* [2, B]

In particolare l'ambiente Gnome, definisce un sistema di interscambio dati basato su un'implementazione di Corba.

3 Corba in Gnome [1, B]

3.1 IDL: Interface Definition Language

Come in tutte le implementazione Corba, dall'Idl si generano client stub e verver skeleton, tramite il comando *orbit-idl* (o in Gnome 2 *orbit-idl-2*).

Il linguaggio di programmazione associato a Gnome è il C e quindi si perdono i vantaggi di altri linguaggi ad oggetti, come ad esempio C++ e Java (in particolare l'ereditarietà).

3.1.1 Mapping di metodi e attributi

I metodi devono essere completamente identificati, quindi per esempio il nome in C del metodo **eat_me** dell'interfaccia **Apple** del modulo **FruitBasket** sarà ***FruitBasket_Apple_eat_me***. Lo standard sconsiglia l'uso di "_" nei nomi e suggerisce di utilizzare le maiuscole, quindi anzichè **eat_me** sarebbe meglio utilizzare **EatMe** e di conseguenza il metodo sarà ***FruitBasket_Apple_EatMe***. Seguendo i canoni di una buona programmazione ad oggetti, gli attributi di un oggetto non sono accessibili direttamente, ma tramite due funzioni: _get-funzione e _set-funzione.

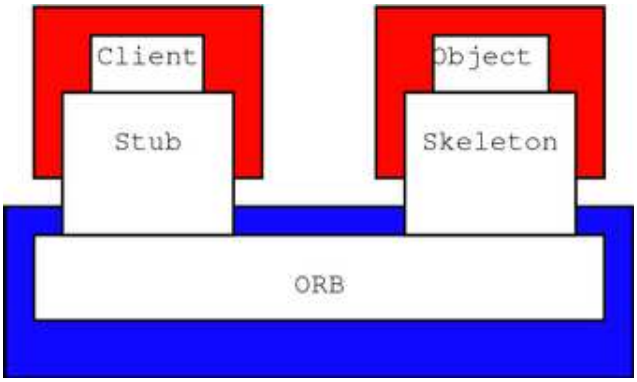

Figura 1: Orb

Errato	Corretto

module FruitBasket {
interface Apple {
attribute boolean is_eaten;
};
};
	module FruitBasket {
	interface Apple {
	boolean _get_is_eaten(void);
	void _set_is_eaten(in boolean in boolean b);
	};
	};

Tabella 1: Accesso equivalente agli attributi di un oggetto (Idl)

```
CORBA_Object app;
CORBA_Environment *ev;
CORBA_boolean bool;
/* codice per connessione al server */
FruitBasket_Apple__set_is_eaten(app, TREU, ev);
bool = FruitBasket_Apple__get_is_eaten (app, ev);
```

Tabella 2: Accesso equivalente agli attributi di un oggetto (C)

Da notare in tabella 2 il doppio underscore tra l'interfaccia ed il metodo.

3.1.2 Mapping dei tipi

tipi IDL	mapping in C
short	CORBA_short
unsigned short	CORBA_unsigned_short
long	CORBA_long
unsigned long	CORBA_unsigned_long
long long	CORBA_long_long
unsigned long long	CORBA_unsigned_long_long
float	CORBA_float
double	CORBA_double
long double	CORBA_long_double
boolean	CORBA_boolean
char	CORBA_char
wchar	CORBA_wchar

Tabella 3: Mapping dei tipi in C

Le costanti ed enum sono mappati con la direttiva *define*, CORBA_boolean è mappato su un char unsigned. Tutti gli altri tipi sono mappati con delle strutture, quindi quando non servono più bisogna liberare la memoria con il comando *CORBA_free()*.

Le sequenze sono mappate con delle strutture:
sequence <char,10> my_var diventerà

```
typedef struct {
    CORBA_unsigned_long _maximum;
    CORBA_unsigned_long _minimum;
    CORBA_char *_buffer;
} CORBA_sequenzce_char;
typedef CORBA_sequenzce_char my_var;
```

3.2 Il modulo CORBA

3.2.1 L'interfaccia CORBA::Object

Ecco una sezione dell'interfaccia CORBA::Object:

```
module CORBA {
interface Object {
        InterfaceDef get_interface ();
        boolean is_nil ();
        Object duplicate ();
        void release ();
        boolean is_a (in string logical_type_id);
        boolean non_existant();
```

```
        boolean is_equivalent (in Object other_object);
        unsigne long hash (in unsigned long maximum);
        }
    }
```

Il metodo is_nil permette di capire se un metodo esiste oppure no, is_equivalent testa se due puntatori sono due istanze dello stesso oggetto. Il metodo duplicate serve perchè molti tipi sono strutture e quindi non è immediato copiarli.

3.2.2 L'interfaccia CORBA::ORB

La prima cosa da fase è inizializze l'ORB, passandogli una stringa univoca che lo identifichi univocamente:

```
#include <orb/orbit.h>

int main (int argc, char **argv)
{
        CORBA_Object orb;
        CORBA_Environment ev;
        orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &env);
};
```

In questo caso la stringa univoca e "orbit-local-orb".

Per fare in modo che degli oggetti possano chiamare i suoi metodi, si può utilizzare un meccanismo semplice: le stringhe IOR. Quando si deve utilizzare un oggetto, la prima cosa da fare è chiamare il server, che gli assegnerà un codice univoco (l'IOR appunto), tramite il quale sarà possibile chiamare l'oggetto. Per accedere a questo dato si usano due metodi di CORBA::ORB: *object_to_string* (chiamata dal sever) e *string_to_server* (chiamata dal client).

3.3 Il Name Service

Il name service in Gnome è implementato dal modulo CosNaming. Il suo compito è quello di associare una stringa leggibile ad un oggetto.

Alcune definzioni utili: il nome è assegnato dal reference dell'oggetto nel contesto dato, risolvere un nome significa trovare il reference del nome nel contesto dato, il contesto è un oggetto CORBA: può essere legato ad un altro oggetto o essere usato per creare un naming graphs. Ogni nodo del naming graph è un contesto e le foglie sono gli oggetti da utilizzare. Il nome di un oggetto è composto dal nome dei nodi e delle foglie necessari a descrivere il camminto dal naming graph tra la root e l'oggetto a cui collegarsi.

Per accedere al name service si utilizzano le sequenti funzioni: *bind, unbind, rebind* (per creare le foglie nel naming graph) e *bind_context, unbind_context, rebind_context* (per creare i nodi nel naming graph).
Ad esempio: *CosNaming_NamingContext_bind (root, &name, panel, &ev)*.

Per inizializzare il name service si usa la funzione di inizializzazione *CORBA_ORB_init*. Inoltre si possono utilizzare due metodi dell'ORB: *list_initial_service* che restituisce una lista dei servizi che l'ORB può risolvere (in CORBA 2.2 sono RootPoa, POACurrent, InterfaceRepository, NameService, TradingService, SecurityCurrent e TransactionCurrent) e *resolv_initial_service* che restituisce il reference ad un oggetto passato come parametro.

3.4 L'interfaccia POA

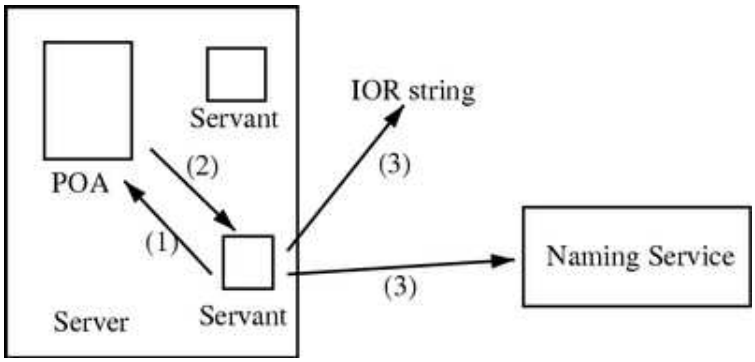

Figura 2: POA

Il Poa è visibile solo dal lato server. Quando un client fa richiesta per utilizzare un metodo, l'oggetto che lo implementa comunica solo con il Poa. L'Orb, il Poa e il server skeleton cooperano per decidere cosa passare al client.

Quando si lancia il server, per prima cosa crea il Poa. Il server dà al Poa informazioni sul servant (passo 1). Questo interroga il Poa per avere un object reference (passo 2) che gli permette di comunicare con il mondo esterno (passo 3) (figura 2).

Le funzioni disponibili per l'interfaccia Poa sono:

- activate_object
- activato_object_with_id
- deactivate_object
- create_reference
- create_reference_with_id
- servant_to_id
- servant_to_reference
- id_to_servant

- id_to_reference
- reference_to_servant
- reference_to_id

Tutte le richieste sono gestite dal PoaManager. Questo si può trovare in quattro stati: *active, inactive, holding o discarding*. Nello stato attivo le richieste sono mandate al Poa, nello stato holding le richieste sono accodate finchè il PoaManager torna allo stato attivo e in discarding mode le richieste vengono ignorate.

Il Poa può avere un certo numero di politiche:

Tipo di politica	**Possibile valore**	**Uso**
ThreadPolicy	ORB_CTRL_MODEL, SINGLE_THREAD_MODEL	Con SINGLE_THREAD, si può usare solo un thread Poa.
LifespanPolicy	TRANSIENT, PERSISTENT	Nel modo TRANSIENT, le implementazioni degli oggetti scompaiono con il porprio POA. Un nuovo Poa sarà creato quando necessario.
ObjectIdUniqueness	UNIQUE_ID, MULTIPLE_ID	Ogni oggetto senza un POA assegnato ha unico ObjectId.
IdAssignmentPolicy	USER_ID, SYSTEM_ID	ObjectId è dato dall'utente o dal sistema.
ServantRetentionPolicy	RETAIN, NON_RETAIN	Nel modo NON RETAIN, a request to a servant already processing another request will throw an ObjectNotExist exception.
RequestProcessingPolicy	USE_ACTIVE_OBJECT _MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER	Ogni richiesta per un servant che non si trova nella mappa del Poa sarà reindirizzata al servant di default e si genererà un'eccezione ObjectNotExist o sarà eseguita una richiesta al Servant Manager per creare il servant richiesto.
ImplicitActivationPolicy	IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION	Se si usa NO_IMPLICIT_ACTIVATION il programmatore dovrà attivare tutti i servant creati.

Tabella 4: Poa policies

3.5 La libreria Gnorba

Gnorba è principalmente un wrapper costruito attorno a due funzioni corba *CORBA_ORB_init* e *CORBA_ORB_resolve_initial_service* ed un'implementazione di un Activation Directory.

Per inizializzare la libreria si usa *gnome_CORBA_init*. Il naming server viene inizializzato con la chiamata *gnome_name_service_get ()*.

3.5.1 GOAD

Il Goad permette di ottenere un elenco dei CORBA sever usabili ed eventualmente lanciarli. Le funzioni disponibili sono:

- goad_server_list_get
- goad_server_list_free
- goad_server_activate
- goad_server_activate_with_id
- goad_server_activate_witch_repo_id

Per aggioungere un server al Goad, bisogna inserirlo nel file *.gnorba*. La sintassi da utilizzare è di questo genere:

```
[gnome_panel]
type=exe
repo_id=IDL:GNOME/Panel:1.0
description=GNOME Panel
location_info=panel
```

3.6 Gnome Desktop

3.6.1 L'interfaccia GNOME::Factory

É usata fondamentalmente come meccanismo di costruzione. Nella programmazione ad oggetti esistono delle funzioni chiamate costruttori, che servono per istanziare gli oggetti. In generale questo è quello che fa l'interfaccia Factory.

Un oggetto dovrebbe incapsulare il suo cotruttore tramite un'implementazione di Gnome::GenericFactory::create_object(). Quindi è possibile chiede all'interfaccia di generare una reference a CORBA::Object dell'oggetto appena creato.

3.6.2 L'interfaccia GNOME::Panel

Lo Gnome Panel ha tre interfacce: GNOME::Panel, GNOME::PanelSpot e GNOME::Applet. Le prime due devono essere implementare dal pannello stesso, mentre l'ultima dall'applet.

L'interfaccia inizializza GNOME::PanelSpot che si occuperà di dialogare con gli applet. In pratica l'applet costruisce un widget e il pannello gli da una finestra in cui disegnarlo. É l'applet che dice al pannello quale menù visualizzare quando viene selezionato con il mouse e quale funzione richiamare. Tutto questo attraverso PanelSpot e Applet.

La prima cosa che deve fare un'applet è verificare se l'interfaccia Applet sta lavorando e se è registrata su un ORB. Quindi deve chiamare la funzione GNOME::Panel::add_applet per dire al pannello che esiste. Il pannello restituirà un unsigned long winid. Il socket widget convertirà il winid con la macro GTK_WINDOW_

XWINOW(widget) e fornirà una finistra in cui lavorare. Plug widget convertirà winid in un GtkWidget. Il pannello potrà creare un socket e restituirà il winid all'applet che ne aveva fatto richiesta, il quale potrà aggiungere il proprio widget al corrispondente widget gtk_plug.

4 Bonobo [1, B]

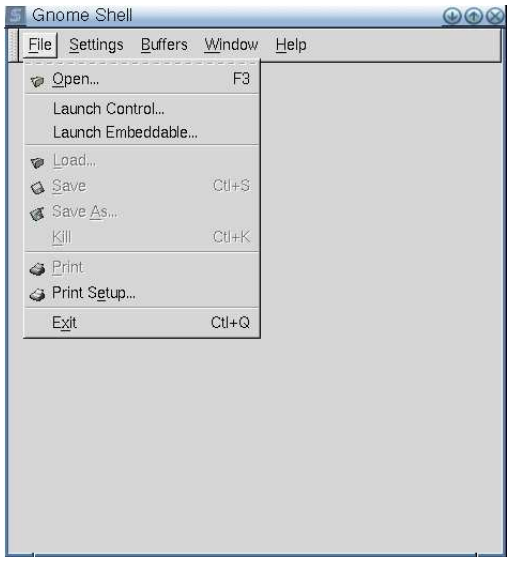

Figura 3: Gshell

Bonobo è un set di interfacce CORBA che implementano alcune i contenitori e altre dei componenti. Inoltre Bonobo è l'implementazione di default di queste interfacce CORBA che vengono esportate attraverso le api in C.

In questo contesto si intende come contenitore un programma in grado incorporare al suo interno dati da un altro programma, che a sua volta si chiamerà componente. Ad esempio il programma *gshell* (figura 3) è un contenitore, in grado di incorporare oggetti esterni. Nella figura 4 c'è un esempio di *controllo* inserito direttamente nel contenitore. Si tratta di una serie di controlli che visualizzano un task list. Nella figura 5, invece c'è un esempio di un componente *embeddable* un documento di testo di Abiword visualizzato all'interno di gshell.

4.1 L'interfaccia GNOME::Unknown

Tutti le interfacce di Bonobo sono ereditate dall'interfaccia GNOME::Unknown.

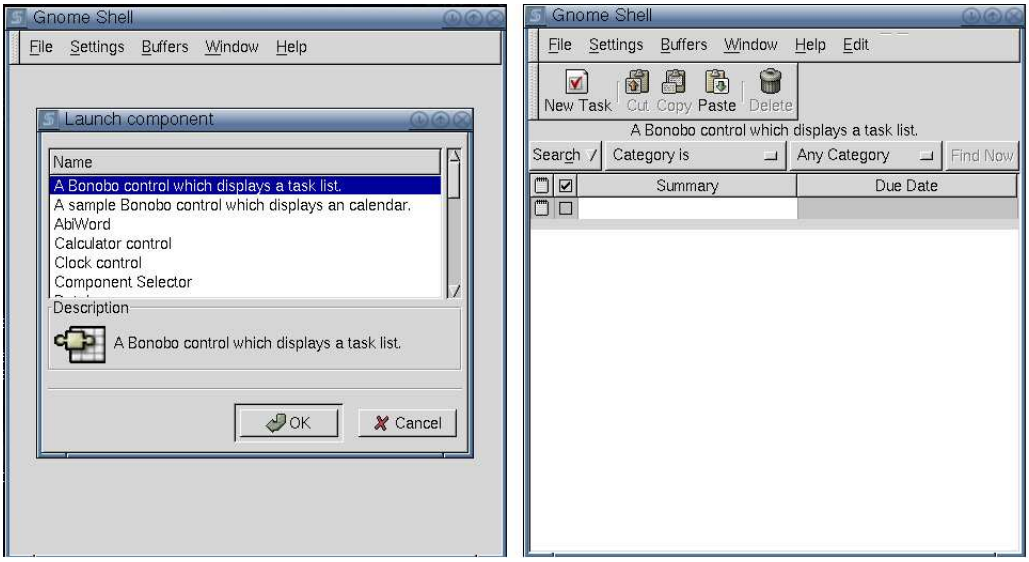

Figura 4: Gshell (1)

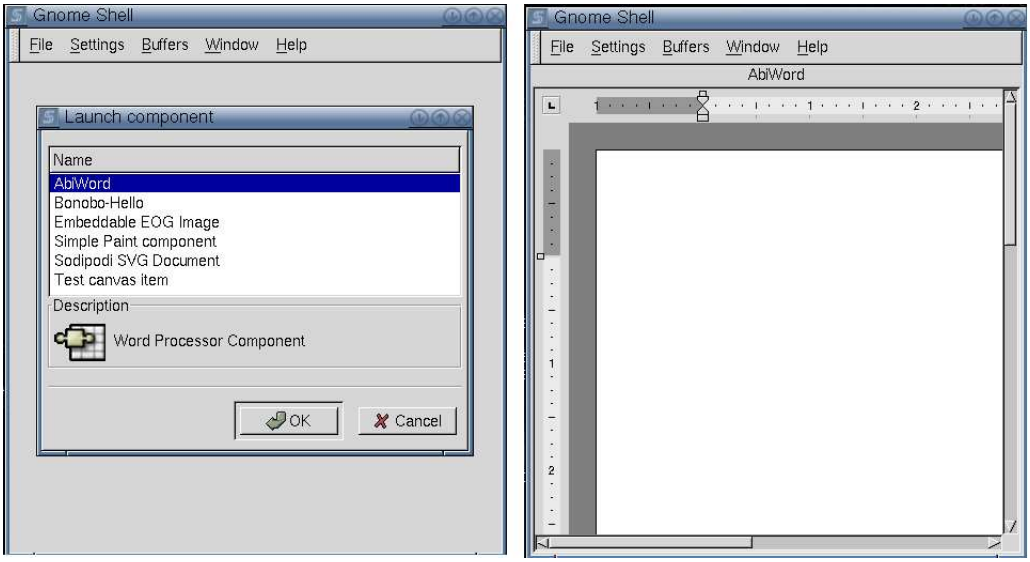

Figura 5: Gshell (2)

4.1.1 Definizione

```
module GNOME {
  interface Unknown {
         void ref ();
         void unref ();
         Object query_interface (in string repoid);
       };
   };
```

La prima delle due funzioni serve per sapere quante funzioni sono linkate all'interfaccia. Questo è necessario per gestire il ciclo di vita dell'oggetto. Sfortunatamente CORBA non ha un meccanismo interno per gestire la morte di un serve. Nel caso succeda i suoi client nel momento in cui cercheranno di contattrlo riceveranno un'eccezione del tipo CORBA_SYSTEM_EXCEPTION. Ogni volta che una funzione si linka ad un'interfaccia viene incrementato ref. Ogni volta che si dereferenzia si usa l'unref. Questo può causare dei problemi nella gestione della memoria: infatti se si eseguono troppi pochi unref si ha un saturamento della memoria (memory leak), mentre nel caso in cui ne vengono fatti troppi si rischia di eliminare un componente premauramente.

Il metodo query_interface è usato per chiedere ad un oggetto quale interfaccia CORBA supporta.

L'interfaccia GNOME::Unknown è implementata attraverso GnomeObject, che a sua volta è ereditato da GtkObject.

4.2 Come cotruire un contenitore

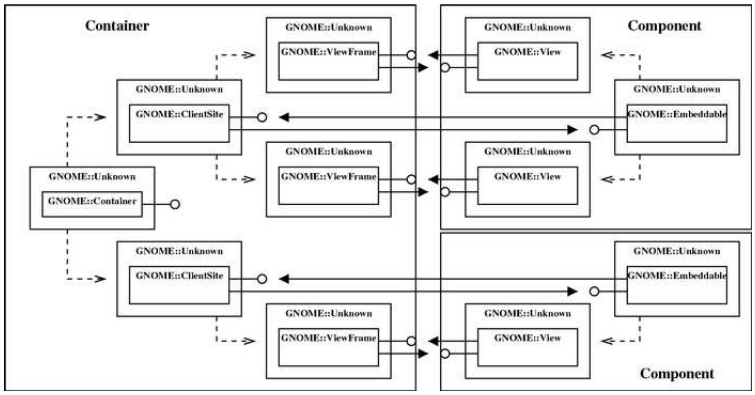

Figura 6: Relazioni tra contenitore e componenti

Un contenitore deve implementare l'interfaccia GNOME::container. Questa è usate per una comunicazione bidirezionale con gli oggetti che implementano GNO-

ME::embeddable. Inoltre il contenitore deve implementare le interfacce GNOME::ClientSite e GNOME::ViewFrame. Se un contenitore supporta l'attivazione "in loco" deve implementare ancora GNOME::UIHandler.

L'idea di base è che quando un contenitore deve incorporare un componente, il contenitore lo aggiungerà alla sua lista dei componenti embedded e creerà ClientSide per il componente e successivamente ViewFrame. Quest'ultima interfaccia dovrebbe essere creata per ogni componente visualizzato.

4.2.1 L'interfaccia GNOME::ClientSite

L'interfaccia contiene:

- la funzione *get_container* è usata per il linking
- una volta che il componente ha notificato al contenitore che deve essere attivato con GNOME::View::activate, si usa il metodo *show_windows* per visualizzare il contenuto della finestra
- il metodo *save_object* serve quando un componente embedded chiede al contenitore di salvare tutte le viste. Il contenitore restituirà Persist::Status che dà al componente l'informazione sull'esito dell'operazione.

4.2.2 L'interfaccia GNOME::ViewFrame

L'interfaccia contiene:

- *request_resize* è un componente di sink: qunado il componente lo chiama chiede al contenitore più o meno spazione se la sua dimensione è cambiata. Il contenitore chiama la funzione GNOME::View::size_allocate della View a cui è connessa GNOME::ViewFrame a cui è stata inoltrata la richiesta.
- anche *view_activate* è un componente di sink: è usato dal componete per essere certo che il contenitore accetti l'attivazione della vista.
- *get_ui_handler* restituisce una reference ad un oggetto che implementa l'interfaccia GNOME::UIHandler. Questa è utilizzata per unire il menu e la toolbar del componente con quelli del contenitore.
- *deactivate_and_undo* dovrebbe essere chiamata dal View durante l'esecuzione se undo è chiamato dopo l'attivazione.
- *get_client_site* restituisce una reference all'oggetto GNOME::ViewFrame dell'interfaccia GNOME::ClientSite.

4.3 Come costruire un componente

Per un componente sono necessarie sono due interfacce: GNOME::embeddable e GNOME::View legate rispettivamente a GNOME::ClientSite e GNOME::View Frame. Un componente dovrebbe implementare l'interfaccia GNOME::PersistFile per supportare il salvataggio.

4.3.1 L'interfaccia GNOME::EmbedableFactory

É l'interfaccia principale del componente Embeddable. Ogni oggetto che lo implementa dovrebbe essere creato con tramite questa interfaccia. Come tutte le interfacce va dichiarata nel file .gnorba come visto nel paragrafo 3.5.1.

- la funzione *create_path* è usata per creare il link
- la funzione *create_object* è usata per creare l'oggetto stesso.

4.3.2 L'interfaccia GNOME::Embeddable

L'interfaccia contiene:

- la funzione *set_client_site* è usata dal contenitore per dire al componente con chi sta parlando. É chiamata ogni volta che il contenitore ha creato il corrispondente ClientSite che permette all'oggetto embedded di cominicare con il contenitore.
- la funzione *set_host_name* è usata per impostare il nome della finestra.
- la funzione *close* è il distruttore del componente embedded
- la funzione *new_view* è la funzione principale di questa interfaccia. É usata dal contenitore per richiedere una nuova vista al componente.
- *get_verb_list* serve al contenitore per chiedere al componente embedded quale "verbs" supporta. Un verbs è un'azione che il componente embedded può eseguire attraverso GNOME::View::do_verb (come ad esempio pagina giù per un visualizzatore di testo).

4.3.3 L'interfaccia GNOME::View

L'interfaccia contiene:

- *size_query* e *size_allocate* sono usate per negoziare la dimensione dal contenitore. Quando un contenitore deve essere ridimensionato chiede a tutti i processi child che dimensione vorrebbero avere. Quindi alloca l'area corrispondente e notifica il risultato con size_allocate
- *set_window* è usata dal contenitore per dare al componente una finestra in cui disegnare la propria vista.

- *activate* è usata per dire ad una vista di attivarsi.
- *reactivate_and_undo* è chiamata dal contenitore quando viene eseguito un undo dopo che un componente è stato disattivato.
- *do_verb* è principalmente usata per chiedere al componente di eseguire una semplice azione non parametrizzata (come ad esempio pagina giù per un visualizzatore di testo).
- *set_zoom_factor*

4.4 L'interfaccia di salvataggio dati in GNOME

4.4.1 Stores

Permette l'accesso ad un file sul disco. Le funzioni disponibili sono molto intuitive (stream si riferiscono a file e storage a directory):

- create_stream
- open_stream
- create_storage
- copy_to
- rename
- commit
- list_contents
- erase

4.4.2 Stream

Su di uno stream si possono utilizzare le seguenti funzioni:

- read
- write
- seek
- truncate
- copy_to
- commit
- close
- eos (end of stream)
- length

4.5 L'interfaccia Persistent

Ogni applicazione che debba accede ad un media persistende deve supportare almeno una di queste interfacce di Bonobo:

- GNOME::Persistent
- GNOME::PersistentFile
- GNOME::PersistentStorage
- GNOME::PersistentStream

4.5.1 L'interfaccia GNOME::Persistent

É ereditata direttamente da GNOME::Unknown ed è la classe base per le altre Pesistent*. É composta da una struttura *status* e *get_goad_id*

4.5.2 L'interfaccia GNOME::PersistentFile

É l'interfaccia specifica per accedere ai file.

- load
- save
- is_dirty
- get_current_file

4.5.3 L'interfaccia GNOME::PersistentStorage

L'interfaccia contiene:

- load
- save
- is_dirty
- init_new

4.5.4 L'interfaccia GNOME::PersistentStream

L'interfaccia contiene:

- load
- save
- is_dirty
- get_size_max

5 Appendice A: comparazione tra DCOM, Corba, Java/Rmi

DCOM	Corba	RMI
Supporta interfacce multiple per gli oggetti e utilizza il metodo QueryInterface() per navigare attraverso le interfacce. Questo significa che un client proxy può dinamicamte caricare server stubs multipli nei layer remoti in base al numero delle interfacce che sono state usate.	Supporta ereditarietà multipla al livello di interfaccia	Supporta ereditarietà multipla al livello di interfaccia
Tutti gli oggetti si implementano IUnknown	Tutte le interfacce ereditano dal CORBA.Object.	Tutti gli oggetti del server implementano java.rmi.Remote
Identifica univocamente gli oggetti di server remoti attraverso il suo interface pointer che serve come object handle in run-time	Identifica univicamente gli oggetti di server remoti attraverso object references(objref), che serve come object handel in run-time. Questi object references possono essere esternati dentro stringhe che possono essere riconvertite in un objref.	Identifica univocamente gli oggetti di server remoti con il ObjlD, che serve come object handle in run-time. Usando .toString() su un riferimento remoto, ci sarà una substringa come "[1db35d7f:d32ec5b8d3:-8000, 0]" che è unica per l'oggetto del server remoto.
Identifica univocamente un'interfaccia usando il concetto di interfaccia Ids (IDD) e identifica univocamente una un'implementazione nominata del server utilizzando il concetto di Class Ids (CLSID) la mappatura di questo si trova nel registro.	Identifica univocamente un'interfaccia usando il nome dell'interfaccia e identifica univocamente un'implementazione nominata dell'oggeto del server in basse alla sua mappatura in un nome nell'Implementatiopn Repository	Identifica univocamente un'interfaccia usando il nome dell'interfaccia e identifica univocamente un'implementazione nominata dell'oggetto del server in base alla sua mappatura in un URL nel Registro
La generazione del reference dell'oggetto del server remoto è eseguita sul protocollo bus dal Ombject Exporter	La generazione del reference dell'oggetto del server remoto è eseguita sul protocollo bus dal Object Adapter	La generazione del reference dell'oggetto del server remoto è eseguita dalla chiamata al metodo UnicastRemoteObject.exportObject(this)
Applicazioni come la registrazione dell'oggetto, skeleton instantation ecc. sono esplicitamente eseguiti dal programma del server o handled dinamicamente dal COM a run-time.	I costruttori eseguono implicitamente le applicazioni comuni come object registration, skeleton instantiation etc.	Il RMIRegistry esegue le aplicazioni comuni come object registration attraverso il Naming class. Il metodo UnicastRemoteObject.exportObject(this) esegue skeleton ed è chiamato implicitamente nel object constructor
Usa la procedura del Opbject Remote Porcedure Call(OSPC) come protocollo remodo di appoggio	Usa il protocollo internet Inter-ORB Protocol(IIOP) come il suo protocollo remodo di appoggio	Usa il protocollo Java Remote Method Protocol(JRMP) come il suo protocollo remodo di appoggio (almeno per adesso)

DCOM	Corba	RMI
Quando un oggetto client ha bisogno di attivare un oggetto server può fare un CoCreateInstance()	Quando un oggetto client ha bisogno di attivare un oggetto server fa un binding su un naming o trader service.	Quando un oggetto cliente necessita un riferimento all'oggetto server deve fare un lookup() sul nome dell'URL dell'oggetto server.
L'object handle utilizzato dal client è il puntatore dell'interfaccia	L'object handle utilizzato dal client è l'Object Reference	L'object handle utilizzato dal client è l'Object Reference
La mappatura dell'Object Name per la sua implementazione è manipolata dal Registro	La mappatura dell'Object Name per la sua implementazione è manipolata dal Implementation Repository	La mappatura dell'Object Name per la sua implementazione è manipolata dal RMIRegistry
Le informazioni sui tipi per i metodi è tenuta in Type Library	Le informazioni sui tipi per i metodi è tenuta nell'Interface Repository	Ogni tipo di informazione è tenuta dall'Object stesso che può essere richiesta usando Reflection e Introspection
La responsabilità di locare un object implementation cade sul Service Control Manager(SCM)	La responsabilità di locare un object implementation cade sul Object Request Broker (ORB)	La responsabilità di locare un Object Implementation cade sul Java Virtual Machine (JVM)
La responsabilità di attivare un object implementation cade sul Service Control Manager(SCM)	La responsabilità di attivare un object implementation cade sul Object Adapter (OA), Basic Object Adapter (BOA) o sul Portable Object Adapter (POA)	La responsabilità di locare un Object Implementation cade sul Java Virtual Machine (JVM)
Lo stub del lato client è chiamato proxy	Lo stub del lato client è chiamato proxy o stub	Lo stub del lato client è chiamato proxy o stub
Lo stub del lato server è chiamato stub	Lo stub del lato server è chiamato skeleton	Lo stub del lato server è chiamato skeleton
Tutti i parametri passati tra gli oggetti client e server sono definiti nel file Interface Definition. Quindi, dipendendo da ciò che specifica l'IDL, sono passati sia in base al valore che in base alla referenza	Quando si passano parametri tra l'oggetto client e l'oggetto server, tutti i tipi d'interfaccia sono passati in base alla referenza. Tutti gli altri oggetti sono passati in base al valore includendo tipi di informazioni altamente complesse	Quando si passano parametri tra l'oggetto client e l'oggetto server, tutti gli oggetti che implementano interfacce estendendo java.rmi.Remote sono passati attraverso una referenza remota. Tutti gli atri oggetti sono passati in base al valore.
Tenta di eseguire garbage collector distribuita sul bus tramite un ping. Il protocollo bus DCOM usa un meccanismo di pinging per il garbage collect remote server object references. Questi sono incapsulati nell'interfaccia IOXIDResolver	Non tenta di effettuare un general-purpose garbage collection distribuito	Tenta di eseguire garbage collector distribuita dell'oggetto server remoto usando il meccanismo contenuto nella JVM

DCOM	Corba	RMI
Può essere eseguito su ogni piattaforma fino a quando ci sarà un COM Service Implementation per la piattaforma stessa. (come Software Ags EntireX)	Può essere eseguito su ogni piattaforma fino a quando ci sarà un CORBA ORB Implementation per la piattaforma stessa. (come Inprise's VisiBroker)	Può essere eseguito su ogni piattaforma fino a quando ci sarà una JVM Implementation per la piattaforma stessa. (distribuita da un grande numero di imprese insieme alla JavaSoft e alla Microsoft)
Poichè la specifica è a livello binario, diversi linguaggi di programmazione come C++. Java, Object Pascal (Delphi), Visual Basic e anche COBOL possono essere utilizzati per codificare questi oggetti	Poichè questa è solo una specifica, diversi linguaggi di programmazione possono essere usati per codificare questi oggetti fino a quando ci sono librerie ORB puoi utilizzare per codificare nel linguaggio scelto	Poichè sono rilasciate pesantemente sul Java Object Serialization, questi oggetti possono essere codificati solo in Java
Ogni metodo chiamante ritorna una struttura well-defined (ben definito) "flat" del tipo HRESULT, i cui bit codificano lo stato di ritorno. Per assurde eccezioni di manipolazione usa Error Objects (del typo IErrorInfo) e l'oggetto server deve implementare l'interfaccia ISupportErrorInfo.	L'Exception handling è controllata dal Exception Object. Quando un oggetto distribuito lancia un oggetto di eccezione l'ORB trasparentemente lo serializza e lo conduce attraverso il bus.	Permettono di lanciare eccezioni che lo serializza e conduce attraverso il bus.

Tabella 5: Comparazione tra implementazioni diverse di componentware [5, W]

6 Riferimenti

I riferimenti con B sono relativi alla bibliografia, mentre quelli con W corrispondono alla sezione web.

Riferimenti bibliografici

[1] Dirk-Jan C. Binnema Diego Sevilla Ruiz, Mathieu Lacage. Gnome & corba. http://developer.gnome.org/doc/guides/corba/.

[2] Salvatore Insalaco. Il drago e la scimmia. *Linux & C*, (numero 11), 2000.

[3] B.C. Stabile M. Pezzuto. Componentware. Tesina di Architettura dei sistemi distribuiti.

Riferimenti web

[1] Java. http://java.sun.com.

[2] Linux. http://www.linux.org.

[3] Microsoft. http://www.microsoft.com/com.

[4] OMG. L'ente che standardizza corba. http://www.omg.org.

[5] Gopalan Suresh Raj. A detailed comparison of corba, dcom and java/rmi. http://my.execpc.com/∼gopalan/misc/compare.html.