Simple Message-Oriented Middleware

Bernardi Giorgio – Matricola 0000.156.516
Università di Bologna
Corso di Laurea Specialistica in Ingegneria Informatica

Abstract

L'interoperabilità di sistemi più o meno complessi risulta essere sempre più frequentemente uno scoglio contro il quale ogni sistemista software deve scontrarsi; ciò a causa dei sistemi hardware a disposizione i quali sono mutati, negli ultimi decenni, da semplici unità centrali di calcolo a reti di elaborazione complesse.

L'articolo tenta di individuare quali siano i componenti essenziali che devono essere forniti ai programmatori per metterli in grado di lavorare con applicazioni diverse senza grossa fatica per poi giungere alla progettazione di una semplice infrastruttura (denominata Simple Message Oriented Middleware, SMom) che funga da middleware fra le diverse applicazioni puntando, in particolare, alla interoperabilità multilinguaggio.

Si giunge, infine, alla progettazione di un servizio per il supporto alla programmazione ad oggetti remota con implementazioni Java e Visual Basic/COM.

Indice

1 Premessa

Spesso è necessario far cooperare applicazioni differenti appartenenti allo stesso sistema o a sistemi diversi. Ciò è vero in molti casi; in via esemplificativa quando si devono aggiungere funzionalità ad un sistema legacy esistente, quando si desidera creare applicazioni residenti su

macchine diverse, quando si desidera sfruttare le potenzialità intrinseche di due linguaggi di programmazione differenti non avendo a disposizione tools che compilino in un unico eseguibile i diversi codici.

Inoltre, capita di voler creare sistemi lasciando ad un secondo momento la decisione del miglior deployment per la soluzione realizzata.

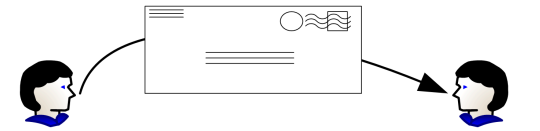
I precedenti problemi possono essere risolti ricorrendo a sofisticati middleware presenti sul mercato; spesso, tuttavia, il costo del middleware, sia in termini finanziari, sia in termini di tempo necessario per il suo setup e per l'acquisizione del know-how necessario per sfruttarlo, ne rendono impensabile l'utilizzo da parte del programmatore.

Il presente articolo tenta di analizzare quali siano i servizi minimi per poter progettare un semplice supporto alla comunicazione fra applicazioni diverse; il sistema sarà inoltre estendibile ad ogni livello, dallo strato applicativo fino allo strato di connessione.

2 Analisi dei requisiti

2.1 Requisiti minimi: comunicazione

Il nucleo di ogni comunicazione è il *messaggio*; per questo motivo anche il nucleo del nostro sistema sarà costituito dall'entità *messaggio* e dalle entità di base che ne consentono l'invio e la ricezione. Il modello che dobbiamo avere in mente è dunque il seguente:



Partecipanti nella spedizione di un messaggio

Il modello prevede 4 entità:

- Il *messaggio*: ovvero il contenuto della comunicazione;
- Il *Sender*: ovvero colui che inizia la comunicazione inviando il messaggio;
- Il *Receiver*: ovvero l'entità alla quale il messaggio deve pervenire;
- Il *canale*: ovvero l'ambiente all'interno del quale il messaggio viaggia per raggiungere il destinatario

Il sistema che vogliamo progettare è distribuito, quindi dobbiamo prevedere che il messaggio non venga consegnato direttamente dal Sender al Receiver; per questo motivo compariranno altre entità le quali hanno il compito di veicolare il messaggio fino alla destinazione.

2.2 Requisiti aggiuntivi

Una volta realizzato il nucleo del sistema, ovvero la comunicazione punto-punto attraverso messaggi, è possibile costruire tanti servizi aggiuntivi. Questi saranno, per quanto possibile, messi a disposizione dal middleware ma potranno, ovviamente, essere realizzati dai diversi utenti finali.

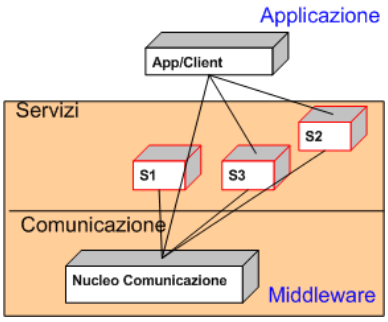
Servizi aggiuntivi tipici sono i seguenti:

- Trasparenza della locazione: uno strumento necessario per il programmatore è senz'altro un sistema che consenta di lavorare con oggetti remoti come se si trovassero localmente; per raggiungere questo obiettivo verranno progettati dispositivi per la costruzione automatica di *Scheleton* e *Stub* che comunicano attraverso il sottosistema di scambio di messaggi.
- Gestione di nomi: sebbene il nostro scopo non sia quello di realizzare un middleware basato su ricerca di oggetti tramite servizi di discovery (in altre parole le applicazioni che cooperano sanno benissimo dove si trovano le risorse di cui hanno bisogno, eventualmente grazie a setup parametrici), è possibile prevedere un servizio in cui le entità Sender e Receiver si registrino per poter essere raggiunte e trovate in maniera

dinamica.
Inoltre è verosimile che il programmatore abbia la necessità di utilizzare nomi simbolici semplici piuttosto che nomi complessi necessari al sistema per identificare tutte le entità: si può quindi pensare di realizzare *servizi di traduzione di nomi*. Infine si potranno realizzare servizi di *discovery* per trovare entità in una rete di grosse dimensioni, di *ticketing* per attribuire nomi univoci agli oggetti e così via.

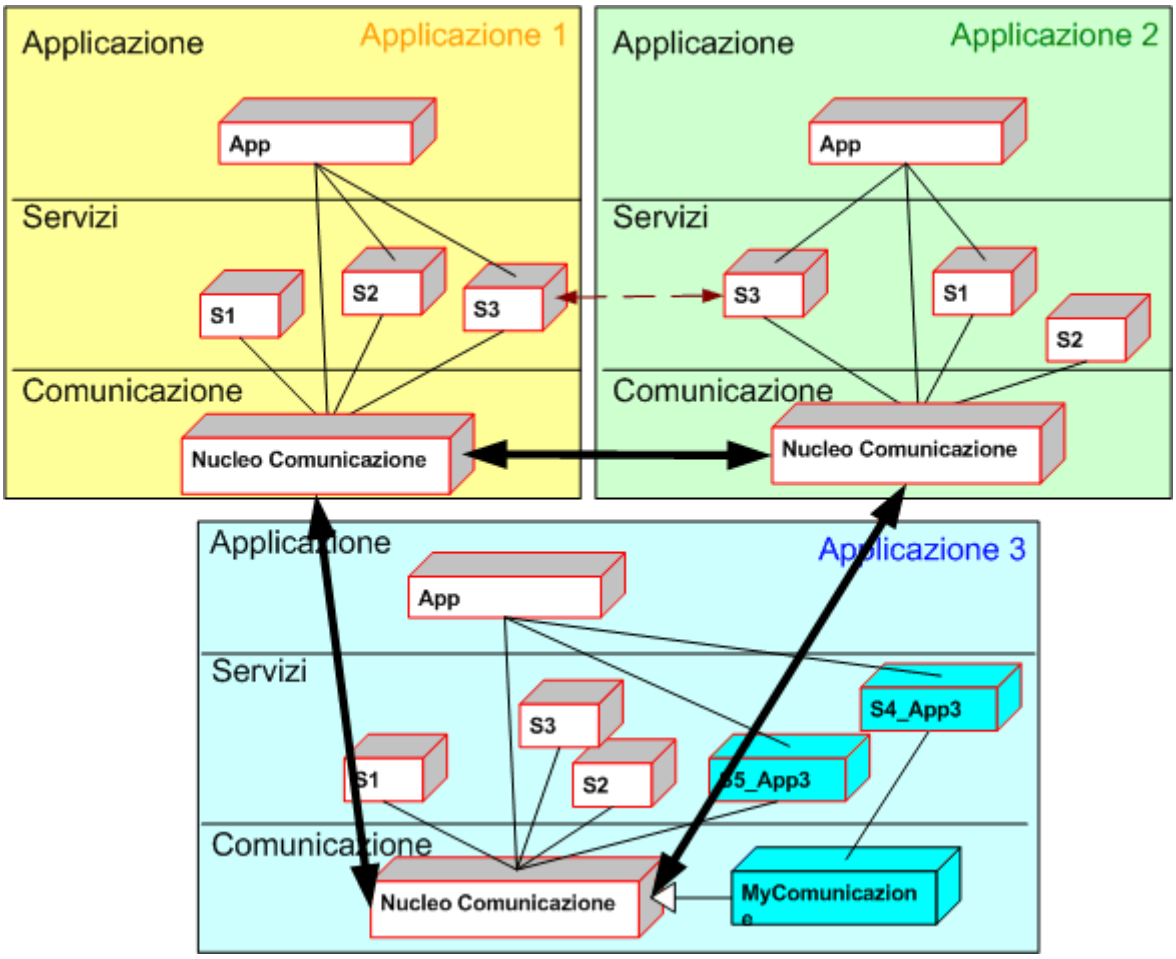
- Garanti della sicurezza: è utile prevedere servizi "chiavi in mano" che consentano l'invio sicuro di messaggi, o a livello di canale di comunicazione, o a livello di codifica del messaggio (cifratura, ecc…)

3 Architettura logica



Come anticipato dai requisiti, il middleware progettato è costruito su (minimo) due livelli: il nucleo di comunicazione e il livello dei servizi aggiuntivi. L'applicazione cliente sfrutta i servizi oppure può accedere direttamente al livello di comunicazione se lo ritiene necessario.

Dal punto di vista dell'intero sistema, avremo diverse applicazioni che cooperano sfruttando i servizi offerti, implementandone di nuovi e comunicando attraverso gli strumenti del middleware.



Architettura di un ipotetico sistema

La figura precedente vuole evidenziare i seguenti aspetti del progetto:

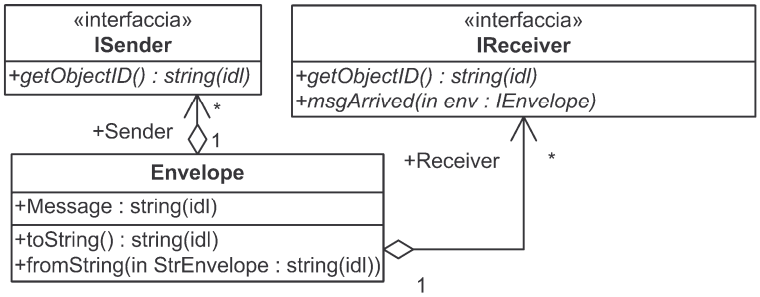
1. Il sistema può essere costituito da più di due nodi di comunicazione punto-punto; *servizi di bridging* potrebbero comunque permettere la connessione di applicazioni non punto-punto.
2. Come nel modello ISO-OSI, i servizi comunicano punto a punto in maniera virtuale (e.g. S3 in app1 e app2) ma la comunicazione fisica avviene sfruttando lo strato sottostante. A questo proposito si utilizza il concetto noto per cui ad ogni livello si impacchetta il messaggio proveniente dal livello superiore in un nuovo messaggio aggiungendo, per esempio, un ulteriore header.
3. L'applicazione 3 mostra come il sistema voglia essere estendibile su tutti i livelli.

4 Progetto

Articoleremo il progetto esaminando le varie parti del middleware cominciando dal nucleo di comunicazione fino ad arrivare ai servizi meno importanti.

4.1 Nucleo di Comunicazione

A livello più basso indicheremo con *Envelope* l'aggregazione di *Messaggio*, *Sender* e *Receiver*. Dal punto di vista dell'*Envelope*, in realtà, questi tre elementi si riconducono a tre stringhe di testo nel senso che per quanto ci riguarda un Sender ed un Receiver sono semplicemente degli oggetti con un identificativo univoco (*ObjectID*). Per quanto riguarda la comunicazione, al contrario, un ricevente dovrà anche essere in grado di gestire i messaggi in arrivo, quindi dovrà prevedere un metodo di notifica. Possiamo mappare queste considerazioni nel seguente disegno.



Progetto dei componenti Envelope, Sender e Receiver

Si noti come, ovviamente, un Sender e un Receiver siano solamente interfacce per il nostro sistema, in quanto nella quasi totalità dei casi Sender e Receiver faranno parte dello strato applicativo in senso stretto.

Oltre alla *Envelope* è necessario progettare l'infrastruttura che permette il passaggio della stessa da Sender a Receiver. Progettando questa infrastruttura sono stati mantenuti suddivisi due aspetti: i meccanismi di comunicazione da un lato e le politiche di gestione e controllo dall'altro. Sono stati dunque progettati due sottosistemi che chiameremo *Dispatcher* e *MsgManager* i quali cooperano per il fine comune.

4.1.1 MsgManager

Un *MsgManager* rappresenta una estremità del canale di comunicazione al livello più basso del nostro middleware. Ogni MsgManager implementa un meccanismo diverso per l'invio dei messaggi nel senso che per ogni canale di comunicazione differente che si desidera utilizzare, si dovrà implementare almeno un MsgManager. Ogni istanza di un MsgManager possiede un nome univoco denominato *Address*. L'*Address* è il corrispettivo dell'*ObjectID* a questo livello di progetto: così come due oggetti si identificano tramite *ObjectID*, allo stesso modo due *MsgManager* si identificano tramite l'*Address*. Ogni implementazione del *MsgManager* potrà utilizzare dei nomi sensati ed utili per il proprio bisogno: per fare un esempio, un manager che utilizza come canale di comunicazione una socket userà come nome qualcosa del tipo *IP:PortaDiAscolto*, mentre un manager che sfrutta un canale di posta elettronica potrebbe avere un nome tipo

casellaPostale@nomeServerPosta. Ogni MsgManager può comunicare con più MsgManager remoti, ma tutti saranno dello stessa classe poiché tutti dovranno ovviamente utilizzare lo stesso canale di comunicazione.

In senso astratto un MsgManager deve possedere un nome univoco (*Address*) e deve offrire una interfaccia standard per la spedizione di una *Envelope*. Tale interfaccia deve prevedere un parametro che indichi l'*Address* del manager di destinazione della busta.

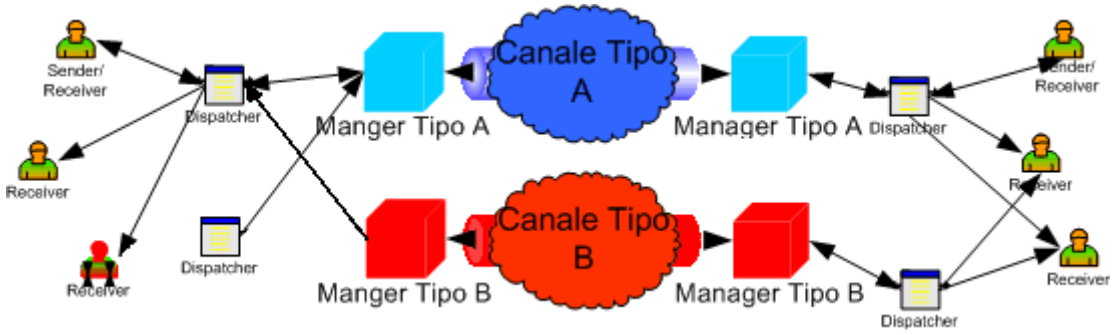
4.1.2 Dispatcher

Un *Dispatcher* è un componente aggiuntivo che si trova dal punto di vista architetturale un gradino sopra rispetto al MsgManager. In altre parole lo strato applicativo e dei servizi dovrebbe colloquiare con un Dispatcher piuttosto che con un MsgManager.

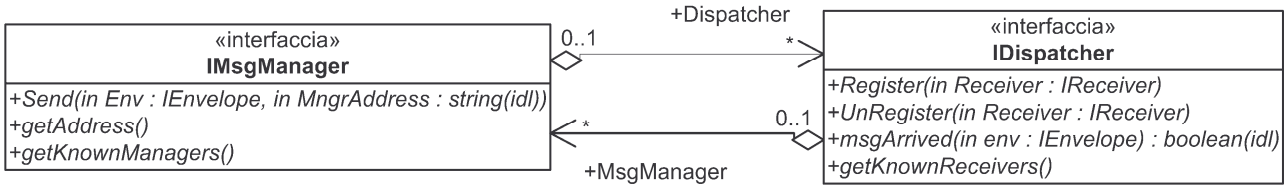
Un *Dispatcher* implementa le politiche di gestione dei messaggi, sia lato mittente, sia lato destinatario. Diciamo che un MsgManager gestisce i messaggi in modo sequenziale e sincrono inviando e distribuendo i messaggi nell'ordine in cui giungono le richieste. Un *Dispatcher*, al contrario, ha lo scopo di permettere al programmatore di usare politiche di spedizione e ricezione differenti. Per esempio si può pensare ad un Dispatcher che consideri N livelli di priorità per i messaggi e mantenga al suo interno N code differenti per inviarli, procedendo prima alla spedizione di quelli più prioritari. Il Dispatcher, poi, può anche implementare politiche differenti rispetto alla sincronia restituendo al cliente il controllo prima ancora dell'invio del messaggio.

Scopo principale del dispatcher durante la ricezione è la distribuzione dei messaggi ai legittimi destinatari. Per far questo un Dispatcher deve consentire ai *Receivers* di registrarsi presso di esso.

Nella figura seguente le frecce indicano la direzione della comunicazione. Frecce bi-direzionali indicano che la comunicazione avviene in entrambi i sensi.



Architettura logica e di funzionamento di *MsgManger* e *Dispatcher*



Progetto dei componenti MsgManager e MsgDispatcher

Le figure precedenti mostrano che un *Dispatcher* può essere legato ad un solo MsgManger in spedizione mentre può essere il referente in ricezione per più MsgManager. Inoltre sia un Dispatcher che un MsgManger possono essere usati in maniera unidirezionale. Ovviamente un Receiver può registrarsi presso più di un Dispatcher per ricevere messaggi e ogni Sender può usare quanti più Dispatcher desidera per inviare le proprie *Envelope*. In pratica il progetto permette moltissime strutture differenti con grande semplicità.

4.2 Componenti per la trasparenza dalla locazione

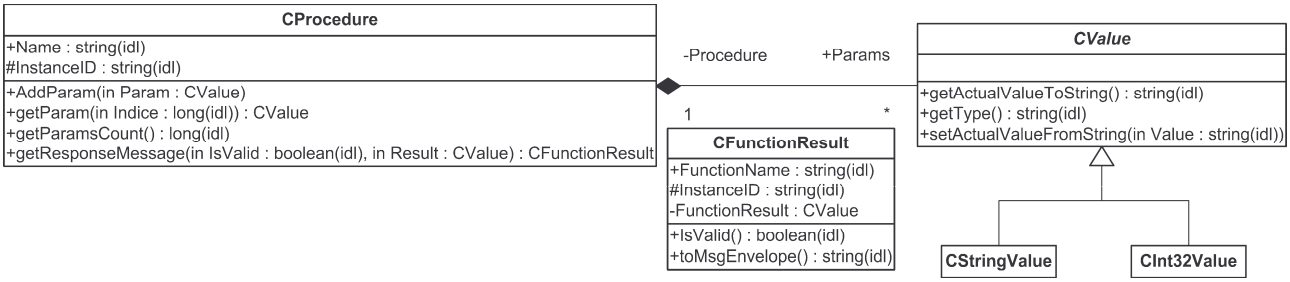
In questa sede si desidera realizzare una infrastruttura di servizio che consenta di lavorare con oggetti locali, facendo in modo che le richieste vengano automaticamente redirezionate verso oggetti remoti. In pratica ipotizziamo di avere una interfaccia nota con una implementazione remota; si desidera realizzare una implementazione locale che dialoghi con quella remota per realizzare i servizi desiderati facendo sì che il cliente non si accorga della differenza (se non per i tempi di esecuzione!).

La struttura progettata ricalca lo schema utilizzato in sistemi analoghi facendo uso di entità aggiuntive di supporto denominate *Stub* e *Skeleton*. Una classe viene vista come collezione di metodi (funzioni e procedure); è stata progettata quindi una rappresentazione di *metodo* che ne consenta l'invio mediante il supporto allo scambio di messaggi SMom. Le funzioni richiedono l'invio di un *risultato*, quindi è stato progettata una rappresentazione che ne permetta l'invio. Inoltre è stato considerato il problema di trasportare l'informazione di *eccezione* generata da un oggetto remoto.

Le procedure senza risultato non prevedono la risposta dal server con un messaggio a meno che non producano eccezioni. Ad un messaggio di richiesta per la chiamata a funzione viene, al contrario, risposto con un messaggio contenente il risultato. Se la procedura/funzione genera una eccezione viene inviato al mittente un messaggio indicante che il risultato non è valido e il valore della risposta consiste in una stringa indicante la descrizione dell'eccezione generatasi.

Ogni chiamata a procedura ha un identificativo generato dal sistema automaticamente in maniera univoca detto *InstanceID*: tale identificativo viene restituito nell'eventuale messaggio di risposta in modo che il Client possa capire a quale chiamata remota il risultato appartiene (siccome lo scambio di messaggi è intrinsecamente asincrono, potrebbero giungere risultati di due chiamate a procedura in ordine diverso da quello di invio; inoltre alcune risposte potrebbero non arrivare a causa di errori).

I componenti progettati hanno la seguente interfaccia:



Struttura degli oggetti per il supporto alla chiamata di procedura remota

I discendenti di *CValue* incapsulano i tipi primitivi previsti dal sistema quali: Stringhe, interi a 32 o 16 bit, booleani, ecc… Ovviamente è necessaria una classe per ogni tipo di dato definito dal SMom.

Chi utilizza il nostro sistema dovrà incapsulare le chiamate utilizzando questi oggetti i quali rendono visibili i metodi per ottenere la rappresentazione in forma di stringa da inviare mediante Envelope con SMom. In particolare lo *Stub* incapsula la richiesta, la invia e attende la risposta; lo *Skeleton* attende i messaggi di richiesta, li decifra, invia la richiesta all'istanza reale della classe recuperando il risultato o una eventuale eccezione, invia allo *Stub* il risultato (o l'eccezione).

Il nostro progetto ha come obiettivo principale quello di velocizzare ed automatizzare il più possibile il processo di sviluppo: per raggiungere questo obiettivo è possibile seguire due strade:

1. progettare dei tools che automatizzino la generazione di Stub e Skeleton statici ad-hoc per la singola classe da usare in remoto;
2. progettare dei componenti standard che permettano di raggiungere una qualsiasi classe in modo dinaimico.

Entrambe le soluzioni sono state considerate.

4.2.1 Tools per l'automazione nella creazione di Stub e Skeleton

Nonostante la letteratura spinga verso l'utilizzo di linguaggi standard per la definizione di interfacce di oggetti (IDL), si ritiene che, per il fine che stiamo considerando, sia migliore una soluzione che prenda in ingresso un file con la definizione dell'interfaccia nel linguaggio in cui è stata definita (e.g. Java, Visual Basic, C++, ecc…) e produca gli *Stub* e gli *Skeleton* (come nel progetto java RMI).

Il tool potrà dare in uscita due tipologie di oggetti:

1. stub e/o skeleton statici che il programmatore potrà poi modificare personalizzandoli con le proprie politiche di gestione (time-out, gestione di eccezioni, ecc…)
2. semplicemente un proxy (utile lato Client) che utilizzi chiamate standard ad uno Stub dinamico standard implementato all'interno del middleware. Lato server sarà al contrario sufficiente lo skeleton dinamico sempre implementato all'interno del middleware

Il tool sarà suddiviso in due parti:

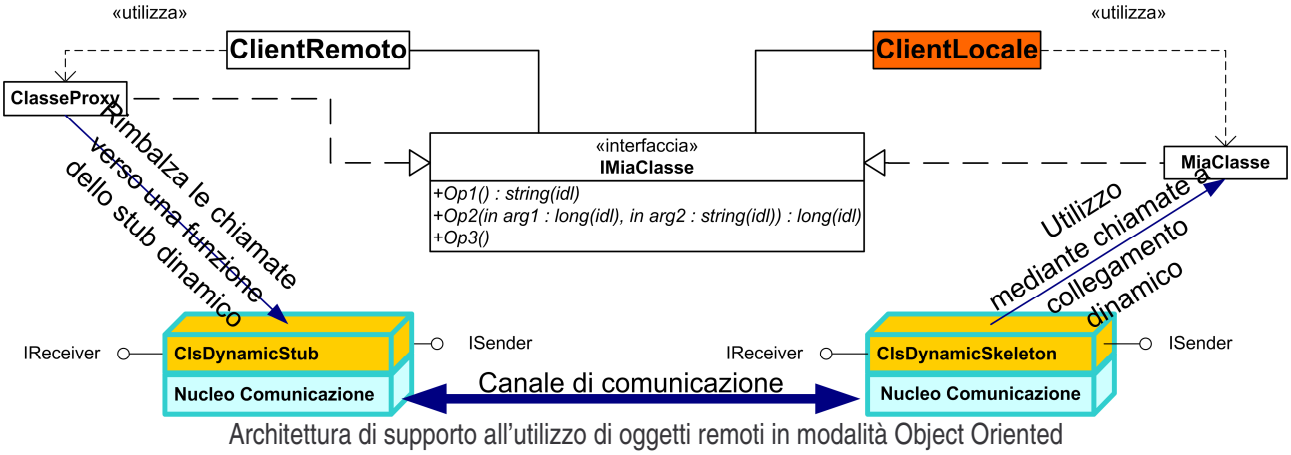
1. l'interprete: in realtà uno o più interpreti (uno per ogni linguaggio di programmazione previsto);
2. il generatore di codice: anche in questo caso uno o più generatori (uno per ogni linguaggio di programmazione previsto) che utilizzano le politiche standard per la generazione.

Dal punto di vista funzionale le operazioni svolte saranno le seguenti.

1. il programmatore scrive la propria classe/interfaccia nel linguaggio A;
2. il programmatore imposta il linguaggio di programmazione di destinazione per stub/skeleton/proxy.
3. il programmatore decide se creare stub/skeleton o proxy.
4. il tool interpreta il file datogli in pasto nel linguaggio A; viene creata una rappresentazione interna dell'interfaccia.
5. il tool analizza i parametri con il quale è stato lanciato e genera i files richiesti nel linguaggio desiderato.

4.2.2 Progetto di componenti Stub e Skeleton dinamici

E' possibile progettare componenti che si adattino dinamicamente a qualsiasi classe in modo da chiamare i metodi effettuando binding dinamici. Tali componenti utilizzano politiche standard in modo che il programmatore debba sviluppare ogni volta poco codice aggiuntivo. L'architettura sviluppata è la seguente. Utilizzando adattatori statici il *Proxy* e lo *Stub* potrebbero essere accorpati.



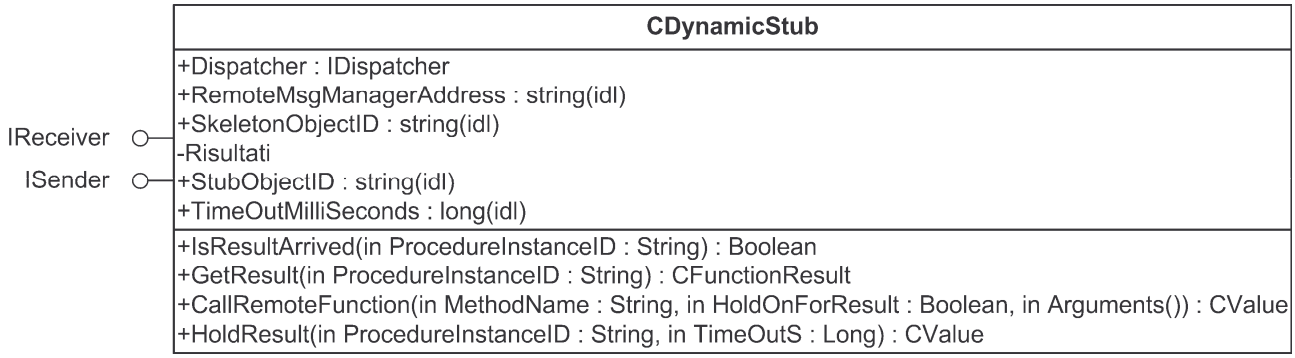
Architettura di supporto all'utilizzo di oggetti remoti in modalità Object Oriented

Lo schema mostra come l'utilizzo dell'oggetto con interfaccia *IMiaClasse* viene utilizzato in maniera tradizionale da Clients locali, mentre un ipotetico Client remoto sfrutta una classe Proxy (da implementare ma generabile in maniera automatica) che si interfaccia con i componenti

dinamici progettati all'interno del Simple Message Oriented Middleware. Il canale di comunicazione può essere scelto a piacimento.

I componenti Stub e Skeleton dinamici hanno le seguenti caratteristiche:

4.2.2.1 Progetto ClsDynamicStub



Progetto dello Stub dinamico contenuto nell'SMom.

Lo stub dinamico è caratterizzato da una interfaccia che permette di richiedere l'invocazione remota di un qualsiasi metodo attraverso la funzione *CallRemoteFunction*. Questa innesca il seguente processo:

1. lo stub incapsula gli argomenti della funzione ed il suo nome in un oggetto CProcedure;
2. viene inviato un messaggio tramite il Dispatcher (precedentemente associato all'oggetto) indirizzandolo allo Skeleton di destinazione.
3. se indicato, la procedura attende il risultato mediante la *HoldResult*, altrimenti ritorna subito il controllo al chiamante restituendo l'*InstanceID* della chiamata: tale valore è necessario per recuperare il risultato della computazione.
4. la procedura *HoldResult* gestisce eventuali eccezioni remote generando a sua volta una eccezione locale con la descrizione dell'evento remoto che ha generato l'eccezione.
5. la procedura *HoldResult* attende la risposta fino allo scoccare di un time-out parametrizzabile

Una classe proxy non deve quindi far altro che chiamare sempre la CallRemoteFunction con i parametri corretti (per questo motivo un proxy è facilmente implementabile in maniera automatica), gestendo le eccezioni remote eventualmente generate. Esempio di pseudo-codice Java-like:

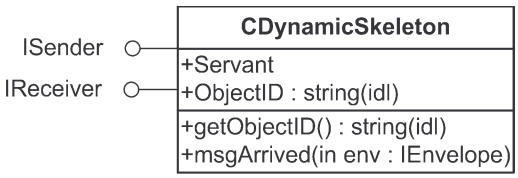
```
public String getValueX(int Index){
	try{
		return DynamicStub.CallRemoteFunction("getValueX", true, Index);
	}catch(RemoteException e){
		//Gestione
	}
}
```

Il progetto così realizzato consente ad un utilizzatore di impostare una propria politica di timeing. Infatti è sufficiente chiamare la funzione *CallRemoteFuncition()* indicando di non attendere il risultato per potersi slegare dalla implementazione di default e andare a recuperare il risultato in modalità polling quando lo si desidera. E' addirittura possibile redirigere il risultato su un altro oggetto che notifichi l'evento di ricezione al cliente, ma non mi dilungherò nell'elenco di tutte le possibili costruzioni che è possibile realizzare usando un po' di immaginazione.

Le funzioni IsResultArrived() e GetResult() sono non-sospensive e permettono di recuperare il risultato in modo polling.

Gli altri parametri della classe impostano semplicemente i nomi necessari per identificare lo Stub come Sender e Receiver all'interno del sistema.

4.2.2.2 Progetto ClsDynamicSkeleton



Lo skeleton dinamico ha una interfaccia molto semplice poiché non necessità di parametrizzazione. Oltre a configurarsi come Sender e Receiver di messaggi (per ricevere le chiamate a procedura e per trasmetterne il risultato) ha la semplice necessità di conoscere l'oggetto *Servant* che implementi i metodi.

Ad ogni messaggio in arrivo compie i seguenti passi:

1. spacchetta il messaggio ricreando l'oggetto *CProcedure*.
2. esegue la chiamata del metodo richiesto per nome (a seconda dei vari linguaggi di implementazione la modalità può essere diversa).
3. se il metodo restituisce un valore o genera una eccezione spedisce un messaggio di risposta al Sender (per poter automatizzare questo processo sono state aggiunte ai MsgManager funzionalità di tracing del messaggio che aggiungono ad una *Envelope* le indicazioni di quali MsgManager locale e remoto hanno gestito il messaggio).

4.2.2.3 Eccezioni

All'interno del sistema sono anche state definite eccezioni standard come le seguenti:

- CRemoteException: eccezione generica lanciata dall'oggetto remoto;
- CTimeOutExpiredException: eccezione generata quando il messaggio di risposta non giunge entro un tempo precedentemente impostato;
- CCannotConvertEnvelopeToCProcedureException: eccezione generata quando non è possibile convertire un messaggio nella Envelope corrispondente;
- Ecc…

4.3 Altri servizi

Altri servizi sono stati pensati ma non hanno raggiunto un livello di progetto tale da essere inseriti in questo articolo.

5 Conclusioni

Come anticipato, scopo del progetto era realizzare un middleware che consentisse l'interoperabilità di ambienti di programmazione e linguaggi di programmazione diversi. Framework quali .NET, DCOM, RMI permettono già una interoperabilità ma è difficoltoso trovare strumenti per l'interoperabilità di Java e COM.

Il sistema progettato è stato implementato in Visual Basic e in Java; sebbene il risultato sia ancora in fase prototipale, è stato raggiunto l'obbiettivo desiderato: realizzare una applicazione Object Oriented che utilizzi oggetti Java "comandandoli" da Visual Basic.

Non sono stati implementati tools per la generazione automatica di codice.

Seguono alcune note che illustrano le scelte implementative "non ovvie" adottate.

Per ulteriori informazioni leggere l'ampia documentazione Html allegata.

5.1 Implementazione Java: package SMom.jar

L'implementazione Java ha fornito un pacchetto jar riutilizzabile da ogni applicazione Java.

Le classi UML sono state mappate in classi Java, i metodi sono stati mappati nei relativi metodi Java, gli attributi sono stati sostituiti da variabili private e da metodi di accesso *set* e *get*.

Sono stati implementati un MsgManager locale ed uno che sfrutta le socket.

Dal punto di vista del supporto alla programmazione remota orientata agli oggetti, è stata sfruttata la Reflection per invocare metodi della classe Servant generica.

Sono state implementate le classi *CStringValue* e *CInt32Value* che incapsulano i tipi di dato *String* e *int* di Java.

Lo stub dinamico sfrutta i thread e la sincronizzazione sulle risorse per permettere le chiamate con attesa del risultato.

5.2 Implementazione Visual Basic/COM: Dll SMom.dll

L'implementazione Visual Basic ha fornito una Dll che espone le interfacce COM riutilizzabile da ogni applicazione che supporta il middleware COM/DCOM.

Le classi sono state mappate in classi Visual Basic, i metodi sono stati mappati nei relativi metodi Vb, gli attributi sono stati per lo più mappati in *Property Get* e *Set/Let*.

Sono stati implementati un MsgManager locale ed uno che sfrutta le socket. Il supporto alle socket è stato realizzato da terze parti.

Dal punto di vista del supporto alla programmazione remota orientata agli oggetti, è stata sfruttata la funzione *CallByName* per invocare metodi della classe Servant generica.

Sono state implementate le classi *CStringValue* e *CInt32Value* che incapsulano i tipi di dato *String* e *Long* di Visual Basic.

Lo stub dinamico non può far altro che attendere con un ciclo per non restituire al cliente il controllo fino all'arrivo della risposta, in quanto Visual Basic non offre un supporto semplice ai thread (anche se qualcosa in tal senso potrebbe essere fatta); per questo motivo il sistema impegna il processore fino all'arrivo del messaggio o fino allo scadere del time-out.

5.3 Sviluppi futuri

Dal punto di vista implementativi sono state tralasciate le definizioni delle altre classi discendenti da *CValue*. Inoltre, è stato rimandato il progetto e lo sviluppo di un impacchettamento sensato dei messaggi inviati attraverso le socket.

Dal punto di vista progettuale (e di conseguenza realizzativo) manca una serie di servizi pensati ma non concretizzati per mancanza di tempo; fra essi dei *MsgManager* che utilizzino canali di comunicazione alternativi quali *mail*, *mailbox* in memoria o su disco, sistemi di tuple memorizzate in database relazionali ecc…