Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Legacy Software Integration Environment:

Metodologie, patterns e tools per l'integrazione nell'ambiente di programmazione Grid.it

Fabio Collura, Saverio Lombardo, Alberto Machì

RT-ICAR-PA-04-15 **Dicembre 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Legacy Software Integration Environment:

Metodologie, patterns e tools per l'integrazione nell'ambiente di programmazione Grid.it

Fabio Collura, Saverio Lombardo, Alberto Machì

Deliverable II° anno
Progetto MIUR FIRB Grid.it
Work Package 8 High Performance Component-based Programming Environment

Rapporto Tecnico N.: **Dicembre 2004**
RT-ICAR-PA-04-15

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

1 Indice

2 Introduzione

Questo documento riassume lo stato di avanzamento del lavoro svolto durante il secondo anno di attività del progetto FIRB ***Grid.it*** **,**dalla Unità di Ricerca ICAR-CNR di Palermo nell'ambito del Workpackage Ambiente di Programmazione.

Scopo generale della ricerca è lo sviluppo di metodologie e strumenti software per l'integrazione di codice legacy nell'ambiente di programmazione ***Grid.it*** Durante il secondo anno ci si è concentrati sulle seguenti linee di intervento:

- metodologie per l'integrazione di codice generico in componente non nativamente sviluppato nell'ambiente di programmazione ASSIST. In questa linea si è definita una metodologia che permette di inglobare codice legacy in un elemento software intermedio (modulo) successivamente integrabile in componenti conformati in base alle regole di composizione e controllo di framework a componenti standard (CCM, Web Services). L'impiego di questi standard rende il codice integrato interoperabile verso componenti sviluppati nativamente nell'ambiente di programmazione ASSIST.
  E' in corso di progettazione un ambiente software che permetta l'adozione di tale metodologia. Sono già state sviluppate alcune librerie C++ di supporto (RPC, eventi).
- definizione di Modelli e Patterns di supporto alla Qualità di Servizio.
  E' stato identificato un modello di adattività alla griglia basata su ruoli e sono stati ideati dei patterns per la gestione di semplici aspetti della Qualità di Servizio applicabili senza intervenire sul codice da integrare.

Nel seguito sono esposti i risultati ottenuti per ognuna delle linee di intervento sopra indicate.

3 Integrazione di codice

Per l'integrazione di codice generico non nativamente sviluppato nell'ambiente di programmazione ASSIST è necessario l'utilizzo di una metodologia rigorosa che, a partire da elementi software di varia natura (codice sorgente, codice oggetto, eseguibili) e realizzati in vari linguaggi di programmazione (C, C++, Fortran), permetta di creare un componente software ospitato in un framework di riferimento (CCM [1] o Web Services). Al fine di rendere il più agevole possibile l'adozione di tale metodologia, si sta cercando di progettare un ambiente software che, in maniera automatica e/o assistita, segua l'utente durante il processo d'integrazione. Tale ambiente sarà sviluppato secondo un approccio iterativo in modo da poter implementare ad ogni iterazione un caso specifico del processo di integrazione. Ciò permette di realizzare alcuni degli scenari di integrazione ben specifici con la possibilità di estensioni future.

La soluzione adottata si basa sulla definizione rigorosa di alcuni **semilavorati** da utilizzare come risultati intermedi del processo di integrazione. Il primo passo del processo consiste dunque nell'analizzare l'elemento software da integrare e nel ricondurre tale elemento in uno di questi semilavorati. Una volta ottenuto il semilavorato si procede con la catena di produzione fino ad ottenere un package (**meta-modulo**) contenente librerie e moduli necessari per il travestimento da componente nel framework di riferimento. In seguito sono riportati i risultati ottenuti lungo questa linea che sono suddivisi nei seguenti sotto-task:

- definizione degli elementi software da utilizzare come semilavorati della metodologia;
- librerie di supporto dell'ambiente di integrazione;

3.1 Elementi software del processo di integrazione

Il presente paragrafo tenta di definire in maniera rigorosa gli aspetti architetturali dei semilavorati identificati durante il processo di integrazione. Gli elementi software legacy, prima di essere integrati, devono essere ricondotti in uno dei seguenti elementi:

- libreria di base;
- kernel;
- modulo;

3.1.1 Interfacce

Una **Interfaccia** è un meccanismo attraverso il quale due elementi software possono scambiare delle informazioni. Le interfacce possibili sono riferimenti a variabili globali condivise e riferimenti a procedure. Le interfacce possono essere di tipo **uses** (o esterne) o di tipo **provides**. Le prime sono dei riferimenti a variabili/procedure esterne, utilizzate dall'elemento software. Le seconde sono invece fornite dall'elemento software. Le interfacce **provides** possono essere **funzionali** o **non-funzionali**. Le prime permettono di attivare una funzionalità dell'elemento software. Le seconde permettono l'interazione con l'elemento software per altri fini (introspezione, configurazione). Ogni interfaccia è caratterizzata da una precisa **signature**, ossia dall'insieme ordinato dei tipi di dati che intervengono nell'interfaccia.

3.1.2 Librerie di base

In questa sezione è presentata l'organizzazione delle librerie di base per la creazione di kernel e moduli. Lo sviluppo di una libreria di base, sebbene non indispensabile, favorisce il riutilizzo del codice sviluppato ed allo stesso tempo incrementa il grado di manutenibilità degli elementi software, forzando una modularizzazione del codice. Una libreria di base *mybaselib* è una **libreria-statica**, ossia è individuata da un file ***lib**mybaselib**.a*** contenente il codice oggetto generato dal compilatore in corrispondenza ad ogni singolo file di codice sorgente della libreria.

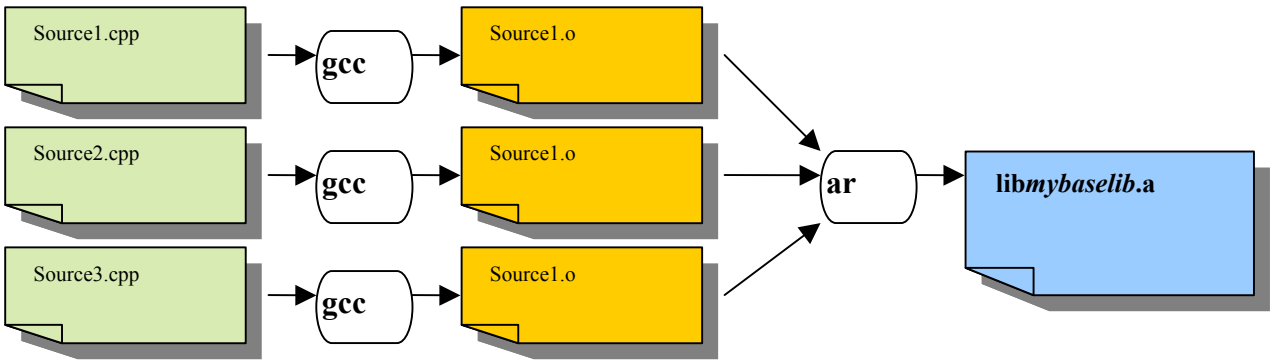

Figura 1. Architettura di un modulo di libreria

Ogni libreria di base ha una **API** (Application Programming Interfaces) di utilizzo, individuata da una collezione di file *header* in cui sono dichiarate le strutture dati, le classi e/o le signature delle procedure implementate nella libreria.

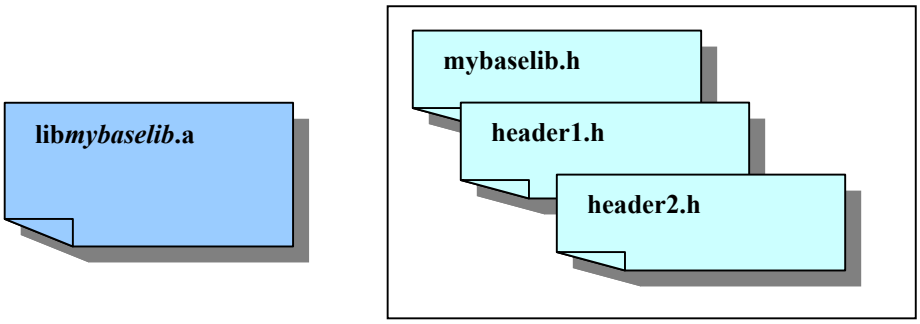

Figura 2. Elementi di una libreria

3.1.3 Kernel

Un ***Kernel*** è un'unità atomica di computazione "in memory" senza stato interno, privo di operazioni di I/O dati ma con meccanismi di output testuale formattato. Un kernel non emette **eccezioni**, né esplicitamente né implicitamente, e termina il suo flusso di esecuzione esattamente al ritorno dalla sua chiamata di attivazione. Qualsiasi comportamento diverso può aversi solamente a seguito di un errore non controllato. Un kernel ha una ***complessità*** dipendente sia da un numero finito di gradi di libertà, ciascuno derivabile da un sotto insieme dei parametri di chiamata, sia, parzialmente, dal valore particolare dei dati da elaborare. Un kernel è normalmente sviluppato a partire da una o più librerie di base con l'intento di uniformare in un'unica chiamata di esecuzione un insieme di attività di uso di elementi della libreria.

La parte funzionale di un kernel è una ***libreria-dinamica***, ossia è individuata da un file ***lib****mykernel****.so*** contenente le librerie di base del kernel. E' possibile corredare un kernel con un metodo non-funzionale che esplicità le variabili indipendenti da cui dipende la sua complessità. Tale metodo è eventualmente presente in un file ***lib****mykernel****-perf.so*** separato, in quanto può essere invocato in un contesto diverso rispetto a quello funzionale.

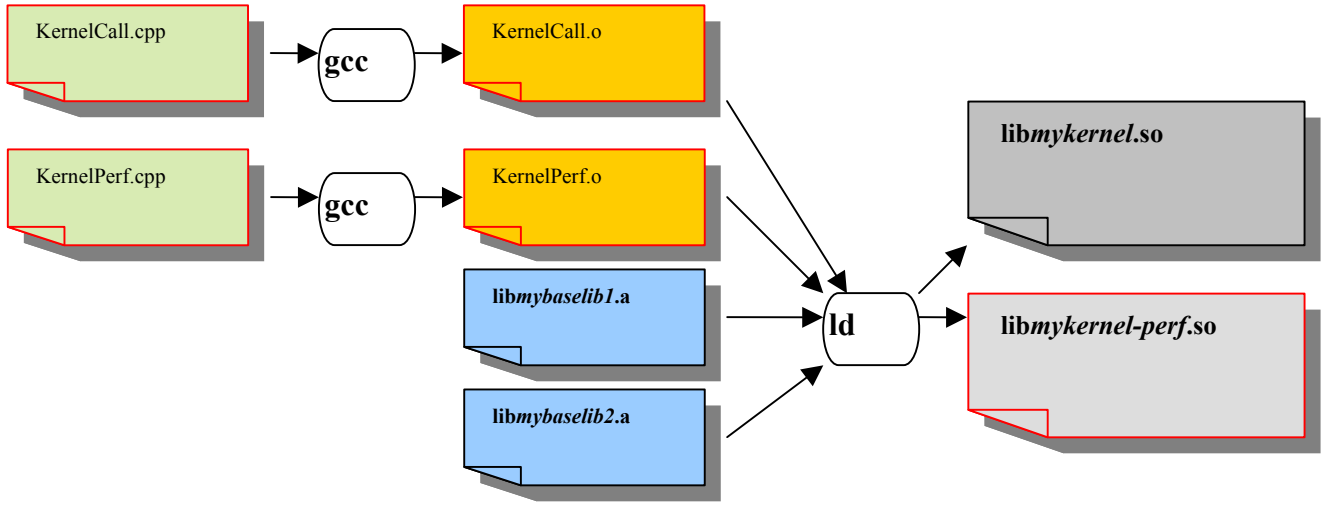

Figura 3. Architettura di un kernel

3.1.3.1 Interfacce funzionali

Un kernel presenta e si identifica in un unico **metodo di chiamata**. Tale metodo ha la seguente interfaccia:

```
extern "C" int kernel-call(param-list);
```

dove *kernel-call* rappresenta il nome della chiamata e *param-list* è la lista dei parametri richiesti:

```
param-list := param | param, param-list
param := param-type param-name
```

Per forzare identificatori univoci del kernel, indipendenti dal compilatore, deve essere utilizzata una simbologia "C". Non è permesso l'overloading funzionale. I parametri di un kernel possono essere di un tipo primitivo (bool, char, int, float, double, …), di un tipo o contenitore standard (string, list, vector, map, …) o di un tipo utente (struct).

I parametri di un kernel possono essere passati per valore (C/C++) o per riferimento (C++/Fortran) se si tratta di tipi primitivi e/o contenitori standard. I parametri di tipo utente devono essere passati per riferimento tramite puntatore. I buffer per i parametri passati per riferimento tramite puntatore devono essere allocati prima della chiamata del kernel. Il kernel non deve restituire un parametro allocato internamente ma può effettuare una riallocazione.

Il kernel termina la sua chiamata restituendo un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine. In particolare, i valori restituiti in caso di errore possono essere:

- -1, EUNKNOWN, errore generico, non specificato o sconosciuto
- -2, EINVAL, valore di un parametro errato

3.1.3.2 Interfacce esterne

Le **interfacce uses** possibili per un kernel sono esattamente due:

```
extern FILE *_stdout
extern FILE *_stderr
```

L'interfaccia *_stdout* rappresenta un riferimento ad uno stream esterno di output testuale formattato. L'interfaccia *_stderr* rappresenta un riferimento ad uno stream esterno di output testuale formattato.

3.1.3.3 Interfacce non-funzionali

Il **metodo di specifica della complessità** di un kernel ha la seguente signature:

```
extern "C" int kernel-perf(
double grade[], int *n, double *datadep, param-list)
```

dove *kernel-perf* rappresenta il nome della chiamata, grade è l'insieme degli *n* gradi di libertà della complessità del kernel, *datadep* è un valore fuzzy relativo alla dipendenza della complessità dal contenuto dei dati d'ingresso e *param-list* è un sottoinsieme dei parametri richiesti per la chiamata funzionale. Il vettore *grade* è inizialmente di dimensione *n*. La chiamata modifica tale parametro *n* secondo il numero effettivo di gradi di libertà. La chiamata restituisce un **valore intero negativo** in caso di errore o di non specifica della complessità, **non negativo** in caso di esecuzione a buon fine.

3.1.3.4 Descrizione

La descrizione a corredo di un kernel deve comprendere tre sezioni principali:

- una **descrizione funzionale**, comprendente tra l'altro la specifica dell'interfaccia di chiamata del kernel e la specifica delle interfacce esterne;
- una **descrizione non-funzionale**, comprendente tra l'altro la specifica della complessità del kernel;

- una **descrizione semantica** generale del kernel;

La descrizione funzionale deve evidenziare l'insieme dei parametri (di chiamata) del kernel e l'esatta dichiarazione dell'interfaccia di chiamata e delle interfacce esterne utilizzate. La descrizione non-funzionale deve evidenziare la cardinalità dell'insieme dei gradi di libertà della complessità del kernel nonché la fuzzyness sulla dipendenza dai dati. La descrizione semantica deve evidenziare, in forma di annotazione testuale, la funzionalità del kernel, il significato dei parametri di chiamata nonché eventuali vincoli e/o limiti di utilizzo, coerentemente con la documentazione rilasciata per il kernel stesso. L'annotazione testuale è di tipo libero.

3.1.4 Modulo

Un **modulo** è un'unità di elaborazione applicativa, ossia è orientato al soddisfacimento di un requisito funzionale di una determinata applicazione o classe di applicazioni, che può effettuare operazioni di **I/O** su risorse di memorizzazione esterne o può richiedere funzionalità esterne per mezzo di **RPC** (Remote Procedure Call).

Un modulo computazionale è un modulo stateless che fornisce metodi per elaborare dei dati applicativi, recuperando e mantenendo i risultati su risorse di memorizzazione esterne.

Un **modulo entity** è un modulo con stato interno che fornisce metodi per memorizzare ed accedere ad una memoria applicativa.

L'interfacciamento di un modulo deve seguire delle direttive al fine di permettere l'integrazione automatica/assistita in un **componente** di un determinato **framework**. Tali direttive possono essere classificate in:

- **requirement**, direttive obbligatorie che devono essere soddisfatte ai fini dell'integrabilità del modulo.
- **recommendation**, direttive opzionali che aumentano il livello di integrabilità del modulo.

Il livello di integrabilità del modulo determina la caratteristiche controllabili e configurabili del componente (I/O, RPC, Grafo di Processamento, Eventi), che incidono sulla flessibilità del pattern di coordinamento applicabile al suo management (Life-Cycle, QoS) attraverso funzioni del framework.

L'insieme delle direttive obbligatorie definisce le specifiche ***minimal compliance*** che garantiscono un ***livello di integrabilità di base***. Tale livello permette la semplice attivazione del

modulo con controllo dell'output testuale ed è sufficiente all'applicazione di un pattern di coordinamento del ciclo di vita del componente.

L'insieme delle direttive opzionali definisce le specifiche ***advanced compliance*** che garantiscono un ***livello di integrabilità avanzata***. Tale livello permette l'attivazione del modulo con configurazione dei ***relevant-data*** necessari all'esecuzione, con possibilità di redirezione dei canali di I/O, con controllo delle chiamate RPC e con meccanismi di gestione di eventi esterni, ed è necessario per l'applicazione di pattern di coordinamento per la gestione della qualità del servizio del componente.

3.1.4.1 Specifiche Minimal Compliance

Le specifiche minimal compliance riguardano l'interfaccia funzionale del modulo e non prevedono interfacce esterne, ad eccezione degli stream di output testuale formattato. Eventuali riferimenti a procedure o variabili globali devono essere risolte in fase di integrazione, come dipendenze a librerie di supporto.

Requirement: Un modulo non emette **eccezioni**, né esplicitamente né implicitamente, e termina il suo flusso di esecuzione esattamente al ritorno dalla sua chiamata di attivazione. Qualsiasi comportamento diverso può aversi solamente a seguito di un errore non controllato.

Requirement: La parte funzionale di un modulo è una ***libreria-dinamica***, ossia è individuata da un file ***lib**mymodule**.so*** contenente le librerie di base e dipendente (a run-time) dalle librerie di supporto (kernel).

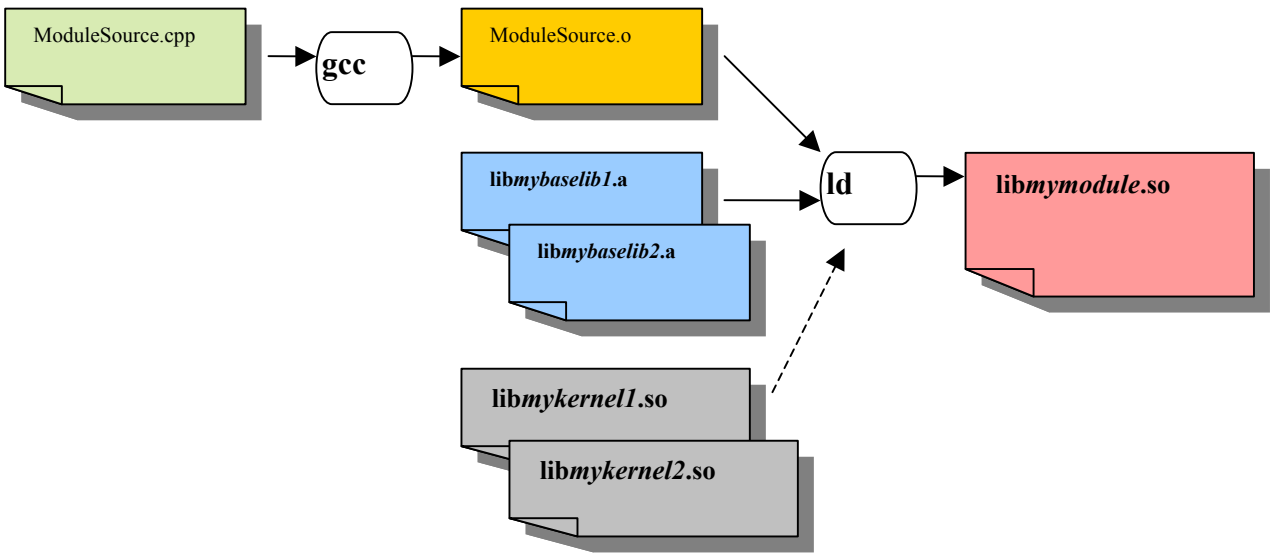

Figura 4. Architettura di un modulo

3.1.4.1.1 Interfacce funzionali

Requirement: Un modulo presenta uno o più **metodi di chiamata**, ciascuno con la seguente interfaccia:

```
int module-call(param-list)
```

dove *module-call* rappresenta il nome della chiamata del modulo e *param-list* è la lista dei parametri richiesti. I parametri di un modulo possono essere di un tipo primitivo (bool,char,int,float,double, …), di un tipo o contenitore standard (string,list,vector,map, ..) o di un tipo utente (struct).

Il modulo termina la sua chiamata restituendo un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

3.1.4.1.2 Interfacce esterne

Requirement: Le **interfacce esterne** previste per un modulo in aderenza alle specifiche ***minimal compliance*** sono esattamente due:

```
extern FILE *_stdout
extern FILE *_stderr
```

L'interfaccia *_stdout* rappresenta un riferimento ad uno stream esterno di output testuale formattato.

L'interfaccia *_stderr* rappresenta un riferimento ad uno stream esterno di output testuale formattato.

3.1.4.2 Specifiche Advanced Compliants

Le specifiche advanced compliants riguardano le interfaccie esterne del modulo per l'accesso alle risorse esterne (I/O) attraverso URL qualificate o per l'attivazione di funzionalità remote (RPC). Eventuali riferimenti a variabili globali devono essere risolte in fase di integrazione, come accesso a risorse esterne di memorizzazione.

3.1.4.2.1 Interfacce esterne

Recommendation: Le interfacce esterne previste per un modulo in aderenza alle specifiche ***advanced compliance*** sono:

```
template <datatype>
extern int _get(const char *uri, datatype *data)

template <datatype>
extern int _put(const char *uri, datatype *data)

extern int _open(const char *uri, const char *mode)
extern int _read(int handle, void *buffer, int size)
extenr int _write(int handle, const void *buffer, int size)
extenr int _ctrl(int handle, const char *mode)
extenr int _close(int handle)
extern int _errno
extern const char *_strerror(int errno)

extern int rpc-call(param-list)
```

L'intefaccia ***_get*** rappresenta un riferimento ad una chiamata esterna di recupero dati strutturati da una risorsa di I/O, parametrizzata in funzione del tipo ***datatype*** dei dati in questione. La risorsa è identificata tramite una **URI/URL**. Il parametro ***data*** contiene il riferimento ad un area di memoria su cui conservare i dati recuperati. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'intefaccia ***_put*** rappresenta un riferimento ad una chiamata esterna di scrittura dati strutturati su di una risorsa di I/O, parametrizzata in funzione del tipo ***datatype*** dei dati in questione. La risorsa è identificata tramite una **URI/URL**. Il parametro **data** contiene il riferimento ad un area di memoria contenente i dati da scrivere. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'interfaccia ***_open*** rappresenta un riferimento ad una chiamata esterna di inizializzazione di una risorsa di I/O. La risorsa è identificata tramite una **URI/URL**. Un parametro **mode** può specificare alcuni attributi della chiamata (read-only, write-only, append, …). La chiamata restituisce un **handle intero positivo** che identifica univocamente la transazione I/O con la risorsa, oppure un **valore intero negativo** in caso di errore.

L'interfaccia ***_read*** rappresenta un riferimento ad una chiamata esterna di lettura dati da una risorsa. La transazione di I/O è identificata tramite il parametro **handle**. Il parametro **buffer** contiene il riferimento ad un area di memoria su cui conservare i dati letti. Un parametro **size** può specificare la quantità di dati da leggere. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'interfaccia ***_write*** rappresenta un riferimento ad una chiamata esterna di scrittura dati su di una risorsa. La transazione di I/O è identificata tramite il parametro **handle**. Il parametro **buffer** contiene il riferimento ad un area di memoria da cui recuperare i dati da scrivere. Un parametro **size** può specificare la quantità di dati da scrivere. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'interfaccia ***_ctrl*** rappresenta un riferimento ad una chiamata esterna di configurazione di una risorsa. La transazione di I/O è identificata tramite il parametro **handle**. Il parametro **mode** specifica gli attributi della configurazione. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'interfaccia ***_close*** rappresenta un riferimento ad una chiamata esterna di finalizzazione di una risorsa. La transazione di I/O è identificata tramite il parametro **handle**. La chiamata restituisce un **valore intero negativo** in caso di errore o **non negativo** in caso di esecuzione a buon fine.

L'interfaccia ***_errno*** rappresenta un riferimento ad una variabile globale esterna per la lettura di un codice in caso di errore da parte di ognuna delle chiamate di I/O.

L'interfaccia ***_strerror*** rappresenta un riferimento ad una chiamata esterna per la lettura di un messaggio testuale relativo all'ultimo errore di I/O o allo specifico errore **errno** passato per parametro.

Ogni risorsa di I/O è identificata attraverso una URI/URL qualificata. Tale qualificatore determina il tipo di risorsa indirizzata.

Per le risorse di memorizzazione, i qualificatori previsti sono:

- ***file***, per l'utilizzo di memorie di massa con accesso diretto
- ***env***, per l'utilizzo di una memoria d'ambiente secondo una modello associativo del tipo ***name*=*value***, dove *name* rappresenta il nome (stringa) della variabile acceduta e *value* rappresenta il valore (stringa) della variabile.
- ***mem***, per l'utilizzo di una memoria applicativa strutturata.

L'interfaccia ***rpc-call*** rappresenta un generico riferimento ad una chiamata di attivazione remota dove *param-list* è la lista dei parametri richiesti, in accordo con l'interfacciamento funzionale di un modulo.

3.2 Librerie di supporto

3.2.1 Remote Procedure Call

E' stata sviluppata una libreria C++ che consente di effettuare delle chiamate remote via socket (Remote Procedure Call) utilizzando un linguaggio standard basato su XML chiamato XML-RPC [2]. XML-RPC è un protocollo di chiamata remota basata su serializzazione dei dati in XML, precursore di SOAP [3], ma più semplice e facilmente aggiornabile ad esso. E' stato scelto come standard intermedio per la prototipizzazione rapida di porte di componenti da utilizzarsi nella sperimentazione dei pattern di integrazione del codice legacy.

Scopo della libreria è fornire delle classi che permettano una creazione automatica degli stubs ed adapters dei moduli legacy al fine di poter invocare remotamente le funzionalità offerte dagli stessi. In figura 5 è riportato un diagramma delle classi della libreria. Le classi principali sono:

- **CommandHandler/CommandSender**: classi che gestiscono la connessione fisica tra due socket. Esse permettono l'invio e ricezione di un flusso di byte tra due end-point.
- **XmlRpcData**: classe contenitore capace di memorizzare qualsiasi tipo di dato primitivo, un dato definito dall'utente, un vettore di dati, o una qualsiasi combinazione tra questi. Questa classe effettua la serializzazione/deserializzazione dei dati in essa contenuti in stringhe XML in base al formalismo dettato dallo schema XML-RPC.
- **XmlRpcUtil**: classe che genera il comando XML che esprime la chiamata remota o il valore ritornato da una chiamata remota.

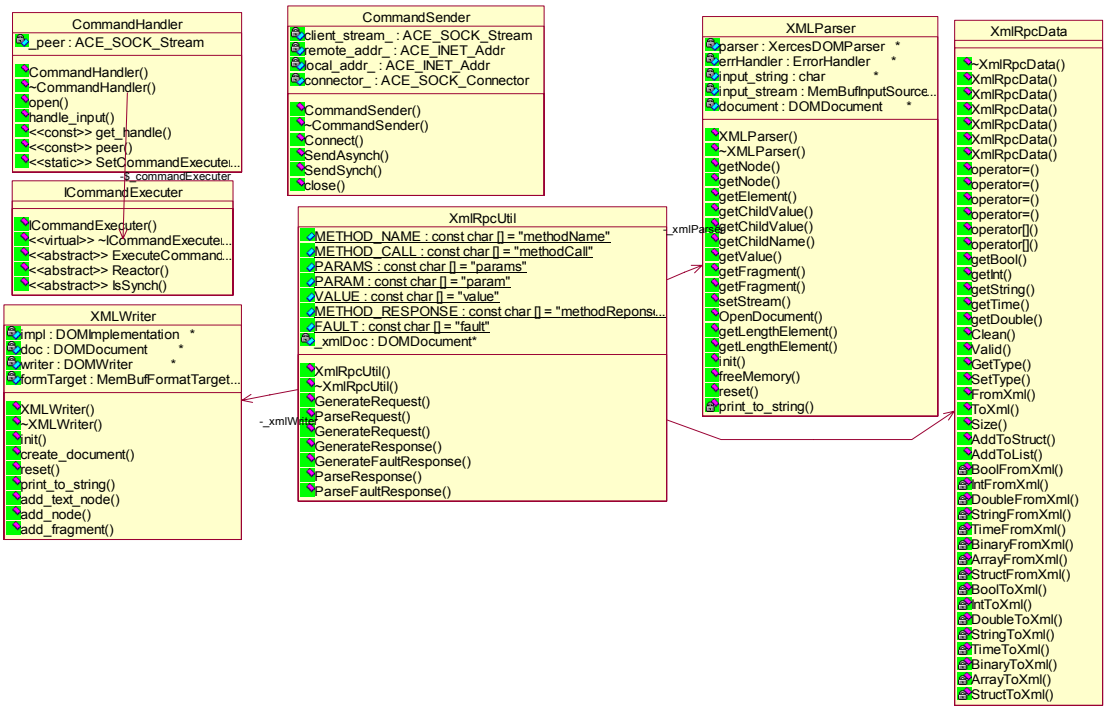

Figura 5. Diagramma delle classi della libreria xml-rpc

3.2.2 Gestione eventi

E' stata sviluppata una libreria che permette di realizzare un bus di comunicazione ad eventi tra elementi software (moduli o componenti). Requisito funzionale del sistema ad eventi è il supporto run-time per l'interscambio di eventi nominati e tipizzati tra componenti di un'applicazione distribuita. Un evento è costituito da una coppia ordinata di due entità tipizzate:

- ***Signal*** (alias *name*), contraddistingue l'evento;
- ***Data***, informazione trasportata dall'evento;

La coppia (signalType,dataType) che contraddistingue ogni evento definisce il ***tipo*** di evento. Gli elementi principali del sistema sono:

- ***Observer***: generico attore interessato alla ricezione di eventi
- ***Subject***: generico attore in grado di emettere eventi;

L'architettura proposta prevede l'utilizzo di un registro centralizzato negli *observer*. Ogni *observer* che desidera ricevere eventi deve registrarsi presso il sistema specificando sia il *tipo* che il *name* degli eventi a cui è interessato. Ogni evento emesso da un *subject* viene ricevuto da tutti gli *observer* registrati per il medesimo *tipo* di evento.

La libreria sviluppata permette di avviare un event-engine locale (di processo) oppure un event-system distribuito (tra processi remoti). L'architettura distribuita prevede la presenza di una entità che operi da *Master* con funzioni di multiplexing degli eventi da e verso i *Client*. Ogni evento emesso da un *client* viene processato dalla event-engine locale (con relativa notifica ai componenti locali) quindi inviato al *Master*. Ogni evento in arrivo al *Master* viene processato dalla event-engine locale (con relativa notifica ai componenti locali) quindi multiplexato a tutti i *client* (eccetto quello di arrivo) registrati per tale evento.

In figura 6 sono riportate le classi principali della libreria, le classi boundary che offrono le API di utilizzo sono:

- **AL_EventEngine:** classe di controllo principale dell'event-engine;
- **AL_Notifier:** classe utility per l'emissione di eventi;
- **AL_Dispatcher:** classe utility per la ricezione di eventi;

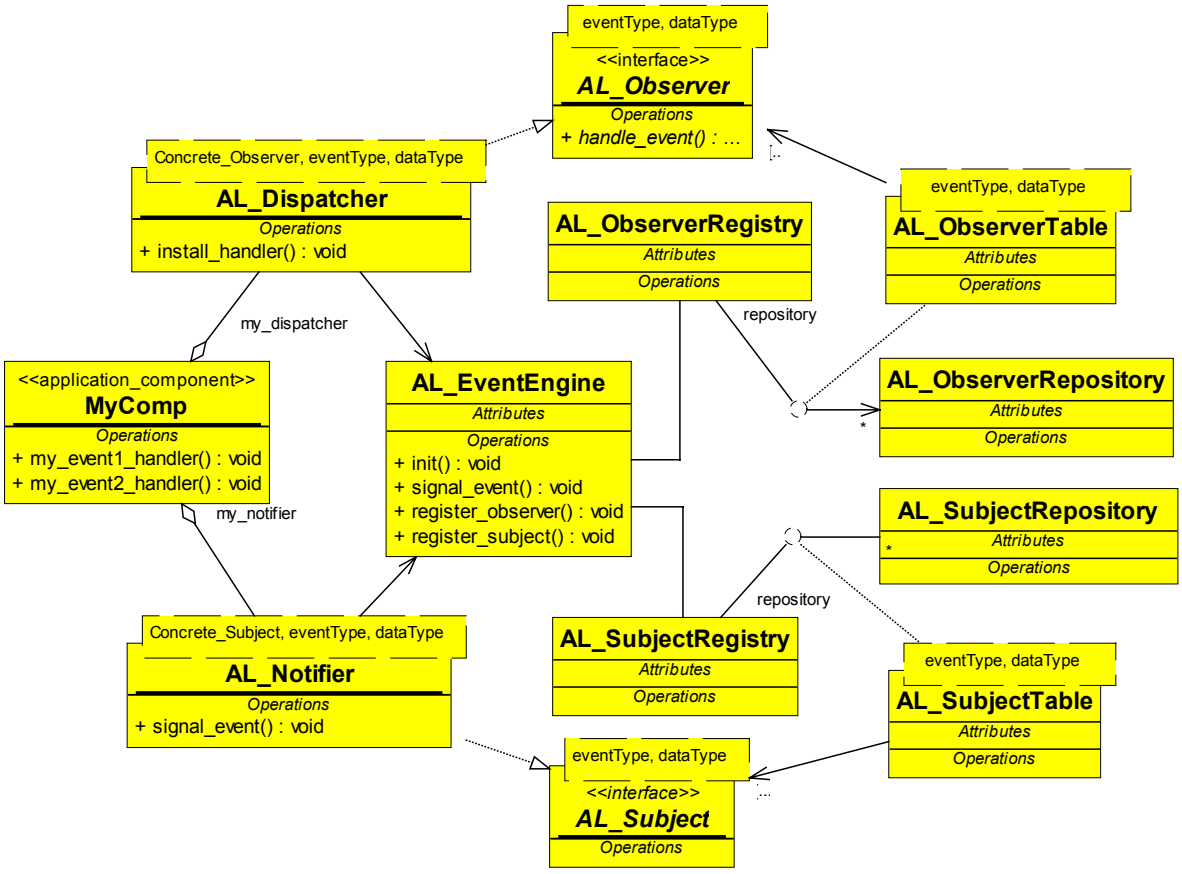

Figura 6. Classi principali della libreria ad eventi

4 Modelli e Patterns di supporto alla Qualità di Servizio

La qualità di servizio (QoS) può essere definita come "l'insieme di quelle caratteristiche quantitative e qualitative che sono necessarie per realizzare le funzionalità richieste di un'applicazione" [4]. Possiamo assumere che la Qualità del Servizio potrebbe essere specificata con un insieme di attributi non funzionali, alcuni dei quali non esprimibili numericamente. È importante specificare che la QoS non è "il livello migliore di queste caratteristiche non funzionali", ma un accordo su un insieme di valori (attributi quantitativi) e su un insieme dei descrittori ontologici (attributi qualitativi) fra le esigenze degli utenti e le qualità specifiche che il fornitore del servizio può offrire.

In un ambiente di griglia l'utente non possiede il controllo diretto sul software, inoltre, la situazione peggiora con il recente interesse verso l'adozione dell'emergente modello Service Oriented Architecture (SOA). Nel modello SOA, infatti, un'applicazione è il risultato del collegamento di vari elementi software (servizi) ognuno dei quali è sviluppato indipendentemente dagli altri per fornire una funzionalità specifica. In tale scenario lo sviluppatore si concentra sull'implementazione delle funzionalità da offrire. Egli non si preoccupa degli aspetti legati alla gestione del flusso esecutivo in cui tale funzionalità sarà inserita. Inoltre, lo sviluppatore non possiede a priori informazioni circa eventuali vincoli o caratteristiche di QoS che saranno richieste durante l'utilizzo del servizio.

Per garantire un certo di livello di QoS il software deve adattarsi ai cambiamenti dell'ambiente in modo da intervenire per correggere situazioni di degrado del livello di QoS concordato. Nel caso di ambienti di griglia, è necessario adattarsi ai cambiamenti di stato delle risorse (grid-awareness or self-adaptiveness) tramite l'attuazione di operazioni non funzionali, quali il *monitoraggio* e politiche di *workflow enforcement*.

Anche se un certo numero di tools e servizi di alto livello sono stati sviluppati dalle comunità di griglia, finora la gestione di tali aspetti non funzionali, come ad esempio la sicurezza del completamento (dependalibility), la fault-tolerance o la gestione delle risorse sono ancora a carico del codice di controllo dell'utente o perfino a carico dell'utente stesso. La più famosa tecnologia di griglia, cioè Globus Toolkit 2 (GT2), come anche la maggior parte dei progetti basati su di essa [5][6], offrono delle primitive (API) sufficienti per lo *staging* dei processi e dei dati e per *l'enactment* dei processi, ma offrono un sostegno limitato

per il *monitoraggio* dei processi, ed appena la primitiva per il *kill* dei processi come supporto al workflow *enforcement*. Il controllo e il monitoraggio dei processi è a carico dell'utente che si trova costretto ad alternare il codice applicativo funzionale con codice dedicato alla gestione degli aspetti di griglia.

Inoltre va aggiunto che, secondo il paradigma delle Organizzazioni Virtuali [7], la gestione delle risorse di griglia non si limita solamente ad aspetti di performance, ma deve tenere conto di considerazioni molto più ampie non direttamente legate alle caratteristiche fisiche delle risorse, come fattori politici o economici, quali ad esempio i costi di utilizzo delle risorse o le politiche di utilizzo dettate dal proprietario della risorsa. Questi aspetti non sono direttamente legati all'applicazione, per tale motivo riteniamo che la adattività alla griglia non dovrebbe essere inglobata interamente nell'applicazione ma parte di essa dovrebbe essere gestita in maniera separata e indipendente dall'applicazione stessa. Per questo motivo abbiamo proposto un modello gerarchico [8] che, applicato ad un servizio di libreria, è capace di fornire moduli software HPC in base ad un certo contratto d'utilizzo sottoscritto con le applicazioni client del servizio. Tale modello ci permette di gestire, in modo gerarchicamente distribuito, aspetti non funzionali dell'applicazione in maniera separata dall'esecuzione delle funzionalità inglobate nei moduli computazionali. Attualmente sono in corso di sperimentazione alcuni aspetti legati alla Qualità del Servizio. In particolare abbiamo modellato dei pattern di gestione dei seguenti (casi semplici ma importanti) attributi di QoS:

- **Dependability**: essa caratterizza il grado di certezza che un'attività sia stata completata o che non sia stata completata [9], questa caratteristica è necessaria in un ambiente di griglia in cui le applicazioni sono eseguite su un insieme di risorse co-assegnate ma non co-riservate.
- **Performance tuning:** nessun meccanismo standard è stato proposto per realizzare il controllo delle prestazioni di skeleton paralleli sulla griglia. Proponiamo un modello di skeleton parallelo master-slave configurabile che coordina l'elaborazione dei dati su una griglia computazionale e fornisce primitive di configurazione al fine di permettere ad un *Manager* esterno di intervenire nel far onorare un certo livello di performance concordato (SLA).

4.1 Un servizio di libreria su griglia basato su ruoli gerarchici

In questo paragrafo descriviamo un modello proposto [8] per realizzare un server di libreria che permetta l'esecuzione remota di componenti software HPC su risorse di griglia. Il server si occupa di amministrazione pro-attivamente sia risorse computazionali che moduli software configurabili per conto di applicazioni esterne ed in base alle strategie di utilizzo dettate da tali applicazioni. Il servizio è in grado di configurare i moduli paralleli per attuare diverse modalità d'esecuzione. Ogni modulo espone un insieme predefinito di modalità di esecuzione (contratti) stabilite in base alle relative possibilità di gestione (monitoring ed enforcement) offerte. Le applicazioni clienti, quando intendono utilizzare remotamente un modulo della libreria, possono effettuare l'ottimizzazione del proprio workflow selezionando la modalità di esecuzione più conveniente (firma del contratto) e delegando il server sul controllo locale (*deployment*, *enactment*, *monitoring*) del modulo la cui amministrazione risulta vincolata dalle specifiche imposte dal contratto sottomesso.

Il modello si basa sull'identificazione di diversi ruoli necessari per realizzare un modello d'adattività alla griglia che sia distribuito tra l'applicazione e l'infrastruttura di griglia. La Figura 7 mostra i ruoli identificati e le relazioni fra essi. Al vertice del sistema giace il ruolo ***attivo*** ricoperto da un **Resource & Execution Manager,** che è il responsabile della gestione delle risorse. Questo ruolo compie funzioni di ricerca delle risorse da acquisire, esegue una selezione delle risorse (nodi e moduli di libreria), e tiene conto delle regole di utilizzo delle risorse acquisite. Al secondo livello risiede il ruolo ***pro-attivo*** del **Resource Administrator**: questo ruolo amministra le risorse in base alle direttive dettate dal ruolo precedente. Esso rappresenta il front-end del servizio di libreria, espone una interfaccia funzionale per poter prenotare e utilizzare un modulo di libreria. Il suo obiettivo è quello di raggiungere un' ottimizzazione globale dell'utilizzo delle risorse amministrate attraverso l'impiego di politiche di scheduling delle richieste e regole di mapping dei grafi di processi sulle risorse fisiche. Inoltre esso si preoccupa del rispetto dei contratti sottoscritti con i clienti e possiede regole di intervento in caso di violazione.

Il modulo di libreria utilizzato dalle applicazioni clienti è inglobato all'interno di un componente ***reattivo*** che, nel caso di moduli paralleli, incapsula uno **skeleton parallelo riconfigurabile** [10]. Tale skeleton, come spiegato in seguito, possiede politiche di distribuzione dei dati e supporta primitive per l'aggiunta o per eliminazione di nodi slave.

Al fine di poter rendere agevole l'operato dell'amministratore è necessaria la presenza di un **coordinatore delle risorse che** si occupa di fornire delle primitive ad alto livello per l'amministrazione delle risorse su griglia, tale componente riveste dunque un ruolo ***passivo***.

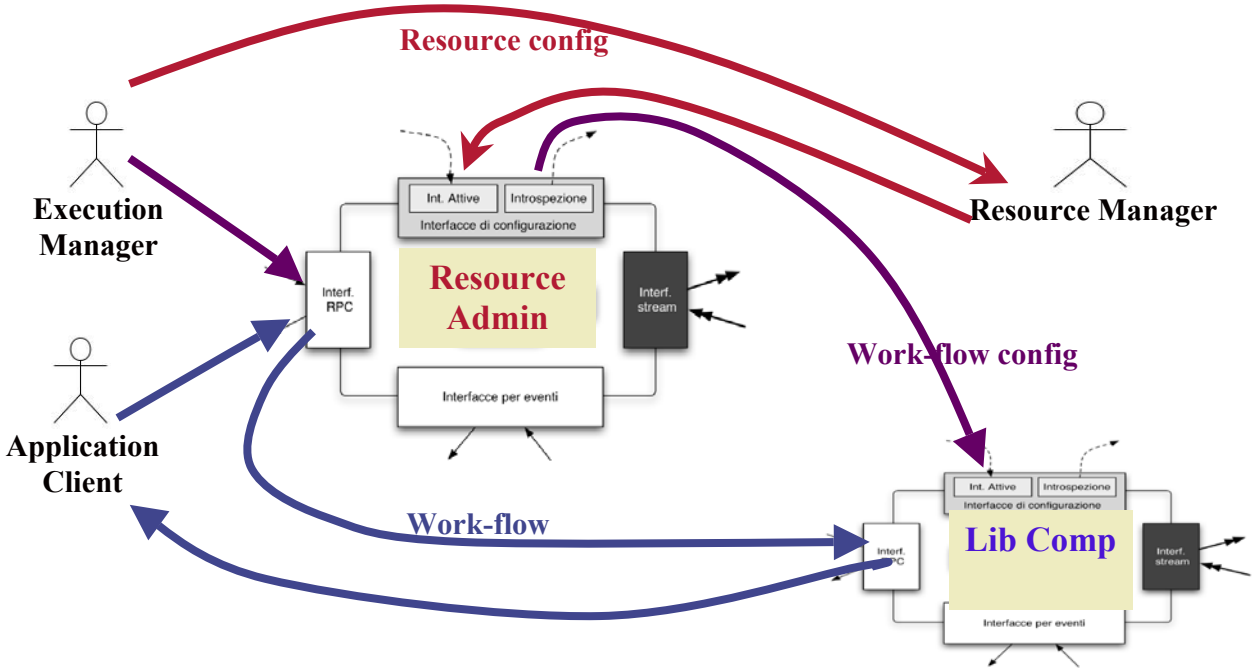

Figura 7. Ruoli e interazioni del modello d'adattività su griglia proposto

Il ruolo del coordinatore delle risorse può essere assegnato ad un middleware di griglia [11], mentre il ruolo di amministratore delle risorse ad una infrastruttura di griglia [8]. Il ruolo attivo di Resource & Execution Manager è lasciato ad attori esterni che possiedono sofisticate capacità di management, come i Problem Solving Environment [12], un Portale di griglia, o un ambiente di programmazione su griglia. Per esempio, l'Application Manager del modello di programmazione di Grid.it [13] potrebbe svolgere tale ruolo. Nella la figura 8 sono indicate le funzionalità principali del ruolo pro-attivo e la sua interazione con gli altri ruoli.

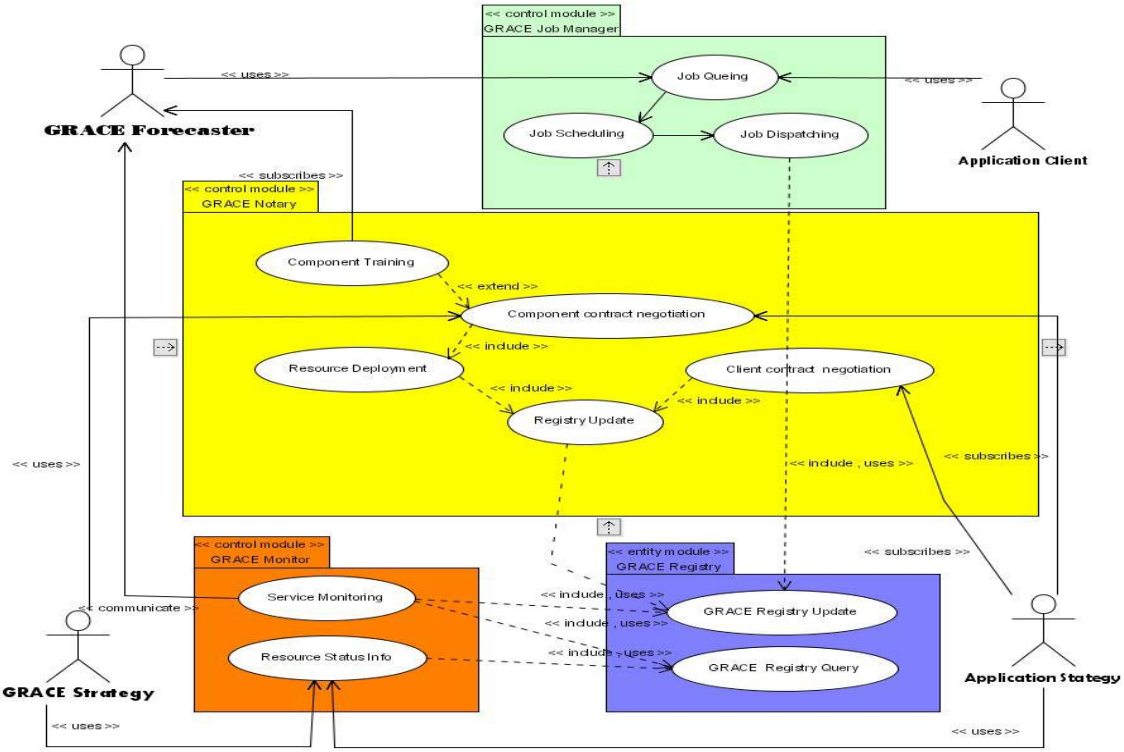

Figura 8. Principali funzionalità del ruolo *proattivo* del Resource Administrator

4.2 Patterns per l'esecuzione sicura su Griglia

In questo paragrafo sono illustrati due pattern che, se applicati per il controllo sull'esecuzione di un grafo di processi generico, permettono di garantire il completamento dell'esecuzione dei processi [14]. Nel caso di fault delle risorse il riavvio dei processi deve essere permessa come operazione minima di ripristino. I due patterns possono essere applicati senza l'intervento nel codice dell'applicazione sfruttando un supporto run-time esterno che offre i servizi collettivi di controllo. Il monitoraggio dell'esecuzione stabile delle attività è eseguito dal ruolo passivo ricoperto dal middleware di griglia. Un processo di *grid-front-end* di tale middleware coordina una rete di *demoni* installati dinamicamente sui nodi di griglia utilizzati ed offre un'interfaccia funzionale sincrona RPC per intervenire sul ciclo di vita di grafo dei processi (deployment, stato di esecuzione, attivazione, kill, etc). I patterns ideati abilitano il *grid-front-end* a controllare sullo stato di esecuzione del grafo dei processi e ad un *Manager* esterno di sapere se un nodo del grafo, o una connessione fra nodi del grafo, è caduta in modo da poter riavviare l'esecuzione del grafo. In questi patterns il *front-end* del middleware comunica con i *demoni* sia sin maniera sincrona (RPC) che in modo asincrono

tramite eventi. Il *Manager* invece comunica col *front-end* in modo sincrono tramite chiamate RPC.

4.2.1 Pattern Resource availability & Connectivity Check

Durante l'esecuzione di un grafo di processi il pattern di **resource availability & connectivity check** verifica ad intervalli regolari sia lo stato d'attività dei nodi che la funzionalità dei collegamenti significativi fra loro. La connettività è esaminata attraverso dei *messaggi di ping* scambiati fra i *demoni* presenti alle estremità del collegamento, se qualche collegamento cade allora viene emesso un segnale che informa il *front-end* sulla caduta della connessione. Inoltre, ogni *demone* emette periodicamente un segnale di *hearth-beat* per informare il *front-end* la sua attuale operatività. In caso di assenza prolungata di tale segnale il *front-end* arguisce una condizione di difetto del nodo, dovuto alla morte del demone, alla caduta della connessione o alla caduta della macchina su cui risiede il demone. Il *front-end*, dopo aver verificato la connessione con la macchina, può asserire se la macchina è caduta, quindi registra una condizione di fault sia del nodo che dei processi lanciati, o se la macchina è raggiungibile, quindi cerca di riacquisire il demone perso per ripristinare lo stato dei processi lanciati. Lo stato dei nodi collegamenti e la connettività tra essi risulta sempre aggiornato nel registro del *grid-front-end* e viene esposto al *Manager* esterno tramite una interfaccia funzionale.

4.2.2 Pattern Insured Completion

Il pattern di **insured completion** abilita il *Manager* a gestire una esecuzione certa di un grafo di processi grazie ad una interazione col *grid-front-end* che lo abilita ad avere una conoscenza costante sullo stato di esecuzione del grafo dei processi. Nel caso di guasto di un nodo il *Manager* può riavviare l'esecuzione del grafo. Ogni *demone* controlla l'esecuzione dei processi sul relativo nodo grazie ai segnali emessi dal sistema operativo locale e informa costantemente il *grid-front-end* che aggiorna lo stato del grafo. I seguenti meccanismi accertano la conoscenza valida dello stato di esecuzione del grafo:

- *segnali attivi locali del sistema operativo*: informano il demone locale circa la terminazione normale o anomala dei processi lanciati. Questi segnali sono comunicati asincronamente al *grid-front-end*.
- *segnale Keep-alive*: è comunicato regolarmente dal *grid-front-end* ad ogni *demone* per indicare di mantenersi in vita.

- *arresto automatico*: un *demone* che non riceve più il segnale di *Keep-alive* arguisce un difetto di collegamento con il nodo del *grid-front-end* o la caduta stessa del *grid-front-end* e quindi si spegne pulendo il nodo (kill di tutti i processi lanciati ed eliminazione dei files)

La fig. 9 mostra il diagramma di sequenza UML relativo ad uno scenario d'uso del pattern in cui il *Manager* ordina di riavviare il grafo dei processi in seguito alla caduta di un processo.

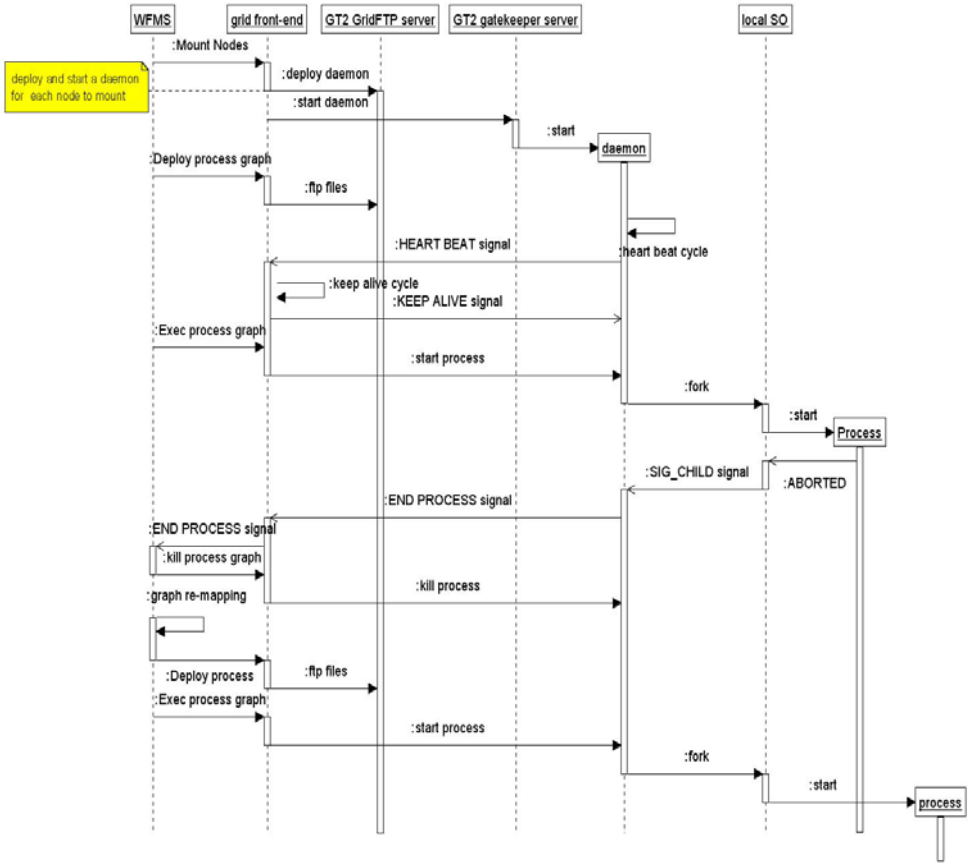

Figura 9. Uno scenario del pattern nel caso in cui il Manager ordina di riavviare il grafo

4.3 Patterns per il controllo sulla performance di Skeleton Paralleli Master-Slave

I patterns e meccanismi illustrati nel paragrafo precedente ci permettono di creare dei pattern più complessi per realizzare un controllo sulla performance di uno skeleton parallelo riconfigurabile [10]. Tale skeleton organizza la politica di distribuzione dei dati sulla base di indici di performance che vengono comunicati dal sistema di management attraverso una operazione di configurazione a tempo di istanziazione. Tale politica mira ad un bilanciamento del carico di lavoro sugli slave ed è basata sulla conoscenza di un indice

caratteristico della potenza effettiva di elaborazione di ciascun nodo slave e di un indice caratteristico della banda efficace del relativo canale di collegamento. Lo skeleton supporta una valutazione della performance da parte di un attore esterno (Workflow-Manager, QoS-Inspector) attraverso l'emissione periodica d'eventi di **CheckPoint** corrispondenti al superamento di alcuni punti di sincronizzazione ed indicanti l'avanzamento dell'elaborazione relativa ai dati dello stream. Lo skeleton supporta delle operazioni di **Enforcement** del Workflow attraverso una modifica del grafo di processo che viene comunicata in maniera asincrona a run-time e diventa operativa soltanto in determinati punti di sincronizzazione, corrispondenti ai CheckPoint. Le primitive di Enforcement sono esattamente due:

- **Slave-Join**, ossia l'aggiunta di un nodo slave al grafo di processamento.
- **Slave-Disjoin**, ossia l'eliminazione di un nodo slave dal medesimo grafo

Entrambe le operazioni di enforcement comportano una modifica nella politica di ripartizione dei dati. Di seguito sono riportati due patterns di interazione che permettono ad un *Manager* esterno di poter controllare il livello di performance durante l'esecuzione dello skeleton.

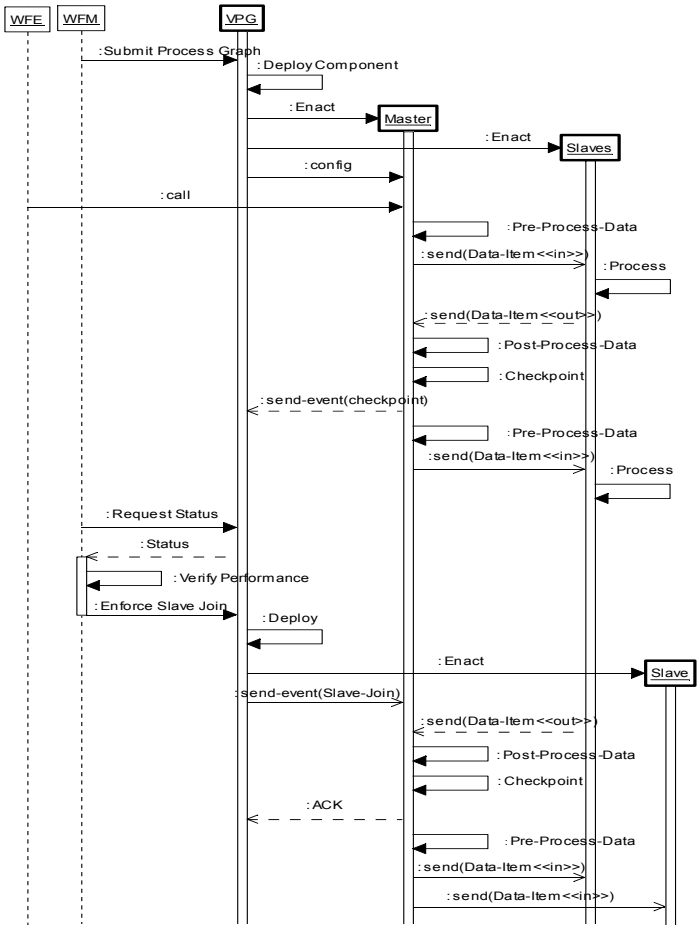

Figura 10. Uno scenario del pattern "performance-tuning"

4.3.1 Pattern Performance-tuning

Il pattern di **performance tuning** permette al *Manager* di calibrare il livello di performance nell'esecuzione dello skeleton variando la cardinalità dei workers. Il diagramma (UML Sequence Diagram) in figura 10 mostra uno scenario nel caso in cui il WFM, a seguito di una analisi dei dati inerenti lo stato di avanzamento dell'attivita, verifica un degrado nella performance di esecuzione del processo parallelo e dispone una operazione di enforcement per l'aggiunta di un worker (Slave Join) al grafo di processamento. L'azione è confermata e diviene operativa al primo superamento del Checkpoint da parte del Master.

4.3.2 Pattern Recover-from-slave-fault

Il pattern di **recover from slave fault** permette allo skeleton di poter proseguire la sua attività anche al verificarsi della caduta di un nodo slave. La figura 11 mostra un diagramma di sequenza UML di un esempio del pattern in una situazione di slave-fault rilevata dalla VPG e notificata al Master tramite una operazione di Slave-Disjoin. Il Master, in attesa di dati sul canale dello slave faulted, conferma l'azione allo scadere di un opportuno timeout,

elimina il work-item in fase di riassemblamento e ripete la fase di distribuzione. Successivamente il pattern di performance tuning è applicato per recuperare il degrado nella performance.

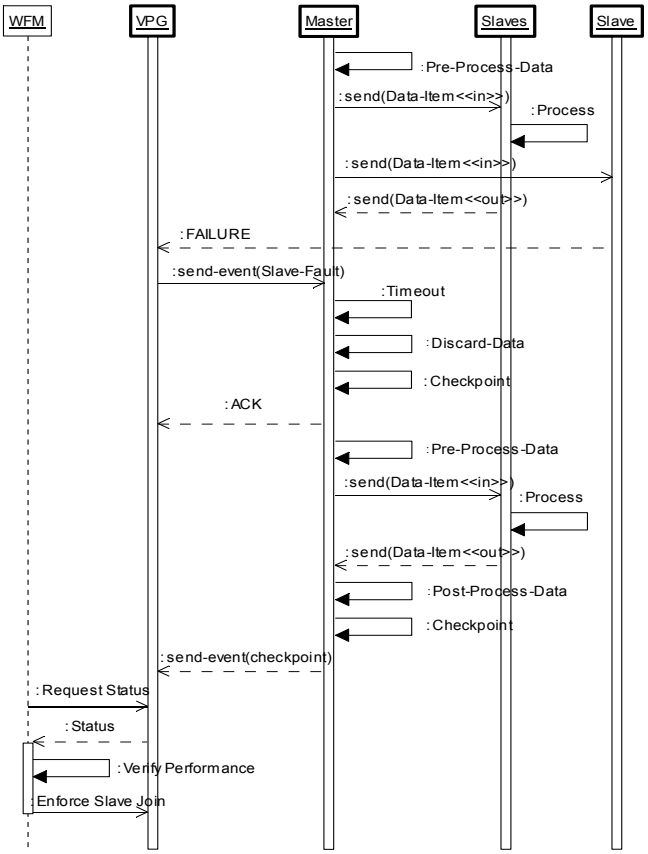

Figura 11. Uno scenario del pattern "recover-from-slave-fault"

5 Riferimenti Bibliografici

[1] OMG. Corba component model, v3.0, November 2001. Document ptc/2001-11-03, available at http://www.omg.org/.

[2] XML RPC Specification, www.xmlrpc.com

[3] Latest version of SOAP Version 1.2 specification: http://www.w3.org/TR/soap12

[4] Vogel A., kerhervé B., Von Bochman G. Ad Gecsei J. (1195). Distributed Multimedia and QoS: A Survey, IEEE Multimedia Vol. 2, No. 2, p10-19

[5] The Condor® Project , http://www.cs.wisc.edu/condor/

[6] The DataGrid Project, http://eu-datagrid.web.cern.ch/eu-datagrid/

[7] The Anatomy of the Grid: Enabling Scalable Virtual Organizations. I. Foster, C. Kesselman, S. Tuecke. International J. Supercomputer Applications, 15(3), 2001.

[8] S. Lombardo, A. Machì. "A model for a component based grid-aware scientific library service". Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, pp. 423-428

[9] ITU-T Rec. I.350: General aspects of Network Performance and Quality of Service in Digital Networks, including ISDN.

[10] A. Machì , F.Collura, S. Lombardo. "A Skeleton supporting Performance Tuning of Grid-enabled Master-slave Components". submitted to 11th International Euro-Par Conference, Euro-Par05, Lisboa Portugal, August 30- September 2, 2005

[11] S. Lombardo, A. Machì: "Virtual Private Grid (VPG 1.2): Un middleware a supporto del ciclo di vita e del monitoraggio dell'esecuzione grafi di processi su griglia computazionale. Versione 1.2.1. Technical report ICAR-CNR- Palermo RT-ICAR-PA 11-2004

[12] M. Cannataro, C. Comito, A. Congiusta, G. Folino, C. Mastroianni, A. Pugliese, G. Spezzano, D. Talia, P. Veltri. Developing a PSE Toolkit for Grids, Technical report ICAR-CNR, Ottobre 2004.

[13] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo. Components for high performance Grid programming in the Grid.it project. In Proc. Of Intl. Workshop on Component Models and Systems for Grid Applications. Held in conjunction with ACMÃ‚Â ICS 2004. Saint-Malo, France, July 2004.

[14] A. Machì., F.Collura, S. Lombardo," Dependable Execution of Workflow Activities on a Virtual Private Grid Middleware", submitted to "2nd International Workshop on Active and Programmable Grids Architectures and Components APGAC'05-ICCS-2005 - Atlanta, USA; 22-25 May 2005

6 Appendice A: Documentazione libreria Eventi

6.1 AL_Dispatcher Class Template Reference

AL_Dispatcher`#include <AL_Dispatcher.h>`
Inheritance diagram for AL_Dispatcher:

6.1.1 Public Types

- typedef void(Concrete_Observer::*** Handler_Type**)(const eventType &event, const dataType &data)

6.1.2 Public Methods

- **AL_Dispatcher** (Concrete_Observer *obs, bool active=false)
- **AL_Dispatcher** (Concrete_Observer *obs, const **AL_LocketSet** &locks, bool active=false)
- void **handle_event** (const **AL_Subject**< eventType, dataType > *subject, const string &pool, const eventType &signal, const dataType &data)
- void **events_handled** (vector< eventType > &events) const
- void **install_handler** (**Handler_Type** handler, const eventType &event)
- void **remove_handler** (**Handler_Type** handler, const eventType &event)

6.1.3 Detailed Description

6.1.3.1 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> class AL_Dispatcher< Concrete_Observer, eventType, dataType >

The AL_Dispatcher allows a concrete observer to install private handler(s) for each event to be observed.
The AL_Dispatcher should be registered to **AL_EventEngine** (*p.33*) as **AL_Observer** (*p.40*).
When event occurr the **AL_EventEngine** (*p.33*) call AL_Dispatcher's event handler which in turn dispatch event to its concrete observer installed handler(s).

6.1.4 Member Typedef Documentation

6.1.4.1 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> typedef void(Concrete_Observer::* AL_Dispatcher::Handler_Type)(const eventType &event, const dataType &data)

The handler type. It's a concrete observer's member pointer.

6.1.5 Constructor & Destructor Documentation

6.1.5.1 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> AL_Dispatcher< Concrete_Observer, eventType, dataType >::AL_Dispatcher (Concrete_Observer * *obs*, bool *active* = false) `[inline]`

Create an AL_Dispatcher to concrete observer.

Parameters:

obs the concrete observer
active is active task

6.1.5.2 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> AL_Dispatcher< Concrete_Observer, eventType, dataType >::AL_Dispatcher (Concrete_Observer * *obs*, const AL_LocketSet & *locks*, bool *active* = false) `[inline]`

Create an AL_Dispatcher to concrete observer.

Parameters:

obs the concrete observer
locks the observer locks
active is active task

6.1.6 Member Function Documentation

6.1.6.1 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> void AL_Dispatcher< Concrete_Observer, eventType, dataType >::events_handled (vector< eventType > & *events*) const [virtual]

Return the event(s) handled list.

Parameters:

events the event(s) handled container

Implements **AL_Observer** (*p.41*).

6.1.6.2 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> void AL_Dispatcher< Concrete_Observer, eventType, dataType >::handle_event (const AL_Subject< eventType, dataType > * *subject*, const string & *pool*, const eventType & *signal*, const dataType & *data*) [virtual]

Handle event (i.e. submit to main processing).

Parameters:

subject the event subject who signal event
pool the event-pool
signal the event-signal
data the event-data

Implements **AL_Observer** (*p.41*).

6.1.6.3 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> void AL_Dispatcher< Concrete_Observer, eventType, dataType >::install_handler (Handler_Type *handler*, const eventType & *event*)

Install handler for event to be observed.

Parameters:

handler the event handler
event the event observed

Todo:

Check duplicate

6.1.6.4 template<class Concrete_Observer, class eventType = string, class dataType = AL_DataSet> void AL_Dispatcher< Concrete_Observer, eventType, dataType >::remove_handler (Handler_Type *handler*, const eventType & *event*) `[inline]`

Remove previously installed handler. (not yet implemented)

Parameters:

handler the event handler to be removed
event the event observed

Todo:

Implementation

The documentation for this class was generated from the following file:

- **AL_Dispatcher.h**

6.1.6.5

6.2 AL_EventEngine Class Reference

AL_EventEngine#include <AL_EventEngine.h>

6.2.1 Public Methods

- void **init** (int argc, char *argv[])
- bool **isClient** ()
  Return true if run as client.
- bool **isMaster** ()
  Return true if run as master.
- bool **isLocal** ()
  Return true if run in local.
- template<class eventType, class dataType> void **register_subject** (**AL_Subject**< eventType, dataType > *subject, const string &pool)
  register subject of given event pool.
- template<class eventType, class dataType> void **register_observer** (**AL_Observer**< eventType, dataType > *observer, const string &pool)
  register observer of given event pool.
- template<class eventType, class dataType> void **signal_event** (**AL_Subject**< eventType, dataType > *subject, const string &pool, const eventType &event, const dataType &data)
- template<class eventType, class dataType> void **send_event** (**AL_Subject**< eventType, dataType > *subject, const string &pool, const eventType &event, const dataType &data)
- template<class eventType, class dataType> void **signal_n_send_event** (**AL_Subject**< eventType, dataType > *subject, const string &pool, const eventType &event, const dataType &data)

6.2.2 Static Public Methods

- AL_EventEngine * **instance** ()
  Return singleton instance.
- template<class eventType, class dataType> void **remoting_register** (const string &pool, const eventType &event)
  remoting register for given event pool.

6.2.3 Detailed Description

The AL_EventEngine manage the event-system.

6.2.4 Member Function Documentation

6.2.4.1 void AL_EventEngine::init (int *argc*, char * *argv*[])

Initialize event-system.

SYNOPSIS: -p <port> the event-stream provide port (default=any) -m <host:port> the event-system master address (i.e. run as client) -l run in local mode (i.e. no event-stream port) -a activate event-stream service

6.2.4.2 AL_EventEngine* AL_EventEngine::instance () [static]

Return singleton instance.

6.2.4.3 bool AL_EventEngine::isClient () [inline]

Return true if run as client.

6.2.4.4 bool AL_EventEngine::isLocal () [inline]

Return true if run in local.

6.2.4.5 bool AL_EventEngine::isMaster () [inline]

Return true if run as master.

6.2.4.6 template<class eventType, class dataType> void AL_EventEngine::register_observer (AL_Observer< eventType, dataType > * *observer*, const string & *pool*)

register observer of given event pool.

6.2.4.7 template<class eventType, class dataType> void AL_EventEngine::register_subject (AL_Subject< eventType, dataType > * *subject*, const string & *pool*)

register subject of given event pool.

6.2.4.8 template<class eventType, class dataType> void AL_EventEngine::remoting_register (const string & *pool*, const eventType & *event*) [static]

remoting register for given event pool.

6.2.4.9 template<class eventType, class dataType> void AL_EventEngine::send_event (AL_Subject< eventType, dataType > * *subject*, const string & *pool*, const eventType & *event*, const dataType & *data*)

Send a named and typed event occurrence.

Parameters:

subject the event subject who signal event

pool the event pool
event the event
data the event data

6.2.4.10 template<class eventType, class dataType> void AL_EventEngine::signal_event (AL_Subject< eventType, dataType > * *subject*, const string & *pool*, const eventType & *event*, const dataType & *data*)

Signal a named and typed event occurrence.

Parameters:

subject the event subject who signal event
pool the event pool
event the event
data the event data

6.2.4.11 template<class eventType, class dataType> void AL_EventEngine::signal_n_send_event (AL_Subject< eventType, dataType > * *subject*, const string & *pool*, const eventType & *event*, const dataType & *data*)

Signal and Send (if possible) a named and typed event occurrence.

Parameters:

subject the event subject who signal event
pool the event pool
event the event
data the event data

The documentation for this class was generated from the following file:

- **AL_EventEngine.h**

6.3 AL_Notifier Class Template Reference

AL_Notifier`#include <AL_Notifier.h>`
Inheritance diagram for AL_Notifier:

6.3.1 Public Types

- enum { **local**, **remote**, **distributed** }
  Notification capabilities.

6.3.2 Public Methods

- **AL_Notifier** ()
  Default Create.
- **AL_Notifier** (const string &info, int mode=local)
  Create with given subject info.
- virtual const string & **info** () const
  The subject class info.
- void **signal_event** (const string &pool, const eventType &event, const dataType &data)
- void **signal_event** (const string &pool, const eventType &event)

6.3.3 Detailed Description

6.3.3.1 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> class AL_Notifier< Concrete_Subject, eventType, dataType >

The AL_Notifier allows concrete subject to signal event. When event occurr AL_Notifier notify all observers registered for that event.

6.3.4 Member Enumeration Documentation

6.3.4.1 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> anonymous enum

Notification capabilities.

Enumeration values:

local
remote
distributed

6.3.5 Constructor & Destructor Documentation

6.3.5.1 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> AL_Notifier< Concrete_Subject, eventType, dataType >::AL_Notifier () [inline]

Default Create.

6.3.5.2 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> AL_Notifier< Concrete_Subject, eventType, dataType >::AL_Notifier (const string & *info*, int *mode* = local) [inline]

Create with given subject info.

6.3.6 Member Function Documentation

6.3.6.1 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> virtual const string& AL_Notifier< Concrete_Subject, eventType, dataType >::info () const [inline, virtual]

The subject class info.

Implements **AL_Subject** (*p.49*).

6.3.6.2 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> void AL_Notifier< Concrete_Subject, eventType, dataType >::signal_event (const string & *pool*, const eventType & *event*)

Signal event occurrence.

Parameters:

pool the event pool
event the event

6.3.6.3 template<class Concrete_Subject, class eventType = string, class dataType = AL_DataSet> void AL_Notifier< Concrete_Subject, eventType,

dataType >::signal_event (const string & *pool*, const eventType & *event*, const dataType & *data*)

Signal event occurrence with carried data.

Parameters:

pool the event pool
event the event
data the event-data

The documentation for this class was generated from the following file:

- **AL_Notifier.h**

6.3.6.4

6.4 AL_Observer Class Template Reference

AL_Observer`#include <AL_Observer.h>`
Inheritance diagram for AL_Observer:

6.4.1 Public Types

- typedef vector< AL_Observer< eventType, dataType > * > **Table**
  A table of AL_Observer.

6.4.2 Public Methods

- virtual void **handle_event** (const **AL_Subject**< eventType, dataType > *subject, const string &pool, const eventType &, const dataType &)=0
  The event handler interface.
- virtual void **events_handled** (vector< eventType > &) const=0
  The event(s) handled list retrieval interface.
- AL_Observer< eventType, dataType > * **observer_interface** ()
  Return this observer interface.

6.4.3 Detailed Description

6.4.3.1 template<class eventType, class dataType> class AL_Observer< eventType, dataType >

The AL_Observer interface allows **AL_Dispatcher** (*p.29*) to be registered as observer for given events occurrence.

6.4.4 Member Typedef Documentation

6.4.4.1 template<class eventType, class dataType> typedef vector<AL_Observer<eventType,dataType> *> AL_Observer::Table

A table of AL_Observer.

6.4.5 Member Function Documentation

6.4.5.1 template<class eventType, class dataType> virtual void AL_Observer< eventType, dataType >::events_handled (vector< eventType > &) const [pure virtual]

The event(s) handled list retrieval interface.

Implemented in **AL_Dispatcher** (*p.31*).

6.4.5.2 template<class eventType, class dataType> virtual void AL_Observer< eventType, dataType >::handle_event (const AL_Subject< eventType, dataType > * *subject*, const string & *pool*, const eventType &, const dataType &) [pure virtual]

The event handler interface.
Implemented in **AL_Dispatcher** (*p.31*).

6.4.5.3 template<class eventType, class dataType> AL_Observer<eventType,dataType>* AL_Observer< eventType, dataType >::observer_interface () [inline]

Return this observer interface.

The documentation for this class was generated from the following file:

- **AL_Observer.h**

6.5 AL_ObserverRegistry Class Reference

AL_ObserverRegistry`#include <AL_ObserverRegistry.h>`

6.5.1 Static Public Methods

- template<class eventType, class dataType> void **register_observer** (**AL_Observer**< eventType, dataType > *observer, const string &pool)
  register observer for given event pool.
- template<class eventType, class dataType> **AL_Observer**< eventType, dataType >::Table * **registered_observers** (const string &pool, const eventType &event)
  Return registered observers for given event.

6.5.2 Detailed Description

The **AL_Observer** (*p.40*) registry manager.

6.5.3 Member Function Documentation

6.5.3.1 template<class eventType, class dataType> void AL_ObserverRegistry::register_observer (AL_Observer< eventType, dataType > * *observer*, const string & *pool*) `[static]`

register observer for given event pool.

6.5.3.2 template<class eventType, class dataType> AL_Observer<eventType,dataType>::Table* AL_ObserverRegistry::registered_observers (const string & *pool*, const eventType & *event*) `[static]`

Return registered observers for given event.

The documentation for this class was generated from the following file:

- **AL_ObserverRegistry.h**

6.5.3.3

6.6 AL_ObserverRepository Class Reference

AL_ObserverRepository`#include <AL_ObserverRepository.h>`

6.6.1 Public Types

- typedef map< string, **AL_ObserverTableInterface** * > **Observer_Tables**
  *A repository of **AL_ObserverTable** (p.45).*
- typedef **AL_Locket**< AL_ObserverRepository > **Guard**
  A repository guard.

6.6.2 Static Public Methods

- void **lock** ()
  Lock repository.
- void **unlock** ()
  Unlock repository.

6.6.3 Friends

- class **AL_ObserverRegistry**
  ***AL_ObserverRegistry** (p.42) can access repository.*

6.6.4 Member Typedef Documentation

6.6.4.1 typedef AL_Locket<AL_ObserverRepository> AL_ObserverRepository::Guard

A repository guard.

6.6.4.2 typedef map<string, AL_ObserverTableInterface *> AL_ObserverRepository::Observer_Tables

A repository of **AL_ObserverTable** (*p.45*).

6.6.5 Member Function Documentation

6.6.5.1 void AL_ObserverRepository::lock () `[static]`

Lock repository.

6.6.5.2 void AL_ObserverRepository::unlock () [static]

Unlock repository.

6.6.6 Friends And Related Function Documentation

6.6.6.1 friend class AL_ObserverRegistry [friend]

AL_ObserverRegistry (*p.42*) can access repository.

The documentation for this class was generated from the following file:

- **AL_ObserverRepository.h**

6.7 AL_ObserverTable Class Template Reference

AL_ObserverTableThe **AL_Observer** (*p.40*) table implementation.

`#include <AL_ObserverRepository.h>`

Inheritance diagram for AL_ObserverTable:

6.7.1 Public Types

- typedef map< string, map< eventType, **AL_Observer**< eventType, dataType >::Table > > **Rep**
  A repository table of AL_Observers.

6.7.2 Public Methods

- **~AL_ObserverTable** ()
- **Rep** & **table** ()
  *The **AL_Observer** (p.40) table reference.*

6.7.3 Static Public Methods

- const char * **typeId** ()
  A unique type ID.

6.7.4 Detailed Description

6.7.4.1 template<class eventType, class dataType> class AL_ObserverTable< eventType, dataType >

The **AL_Observer** (*p.40*) table implementation.

6.7.5 Member Typedef Documentation

6.7.5.1 template<class eventType, class dataType> typedef map<string, map<eventType, AL_Observer<eventType,dataType>::Table> > AL_ObserverTable::Rep

A repository table of AL_Observers.

6.7.6 Constructor & Destructor Documentation

6.7.6.1 template<class eventType, class dataType> AL_ObserverTable< eventType, dataType >::~AL_ObserverTable () [inline]

6.7.7 Member Function Documentation

6.7.7.1 template<class eventType, class dataType> Rep& AL_ObserverTable< eventType, dataType >::table () [inline]

The **AL_Observer** (*p.40*) table reference.

6.7.7.2 template<class eventType, class dataType> const char* AL_ObserverTable< eventType, dataType >::typeId () [inline, static]

A unique type ID.

The documentation for this class was generated from the following file:

- **AL_ObserverRepository.h**

6.8 AL_ObserverTableInterface Class Reference

AL_ObserverTableInterfaceThe interface of an **AL_Observer** (*p.40*) table.

`#include <AL_ObserverRepository.h>`

Inheritance diagram for AL_ObserverTableInterface:

6.8.1 Public Methods

- virtual **~AL_ObserverTableInterface** ()

6.8.2 Detailed Description

The interface of an **AL_Observer** (*p.40*) table.

6.8.3 Constructor & Destructor Documentation

6.8.3.1 virtual AL_ObserverTableInterface::~AL_ObserverTableInterface () [inline, virtual]

The documentation for this class was generated from the following file:

- **AL_ObserverRepository.h**

6.9 AL_Subject Class Template Reference

AL_Subject`#include <AL_Subject.h>`
Inheritance diagram for AL_Subject:

6.9.1 Public Types

- typedef vector< AL_Subject< eventType, dataType > * > **Table**
  A table of AL_Subject.

6.9.2 Public Methods

- virtual const string & **info** () const=0
  The subject class info.
- AL_Subject< eventType, dataType > * **subject_interface** ()
  Return this subject interface.

6.9.3 Detailed Description

6.9.3.1 template<class eventType, class dataType> class AL_Subject< eventType, dataType >

The AL_Subject interface.

6.9.4 Member Typedef Documentation

6.9.4.1 template<class eventType, class dataType> typedef vector<AL_Subject<eventType,dataType> *> AL_Subject::Table

A table of AL_Subject.

6.9.5 Member Function Documentation

6.9.5.1 template<class eventType, class dataType> virtual const string& AL_Subject< eventType, dataType >::info () const `[pure virtual]`

The subject class info.

Implemented in **AL_Notifier** (*p.38*).

6.9.5.2 template<class eventType, class dataType> AL_Subject<eventType,dataType>* AL_Subject< eventType, dataType >::subject_interface () [inline]

Return this subject interface.

The documentation for this class was generated from the following file:

- **AL_Subject.h**

6.10 AL_SubjectRegistry Class Reference

AL_SubjectRegistry`#include <AL_SubjectRegistry.h>`

6.10.1 Static Public Methods

- template<class eventType, class dataType> void **register_subject** (**AL_Subject**< eventType, dataType > *subject, const string &pool)
  register subject of given event pool.

6.10.2 Detailed Description

The **AL_Subject** (*p.48*) registry manager.

6.10.3 Member Function Documentation

6.10.3.1 template<class eventType, class dataType> void AL_SubjectRegistry::register_subject (AL_Subject< eventType, dataType > * *subject*, const string & *pool*) `[static]`

register subject of given event pool.

The documentation for this class was generated from the following file:

- **AL_SubjectRegistry.h**

6.11 AL_SubjectRepository Class Reference

AL_SubjectRepository`#include <AL_SubjectRepository.h>`

6.11.1 Public Types

- typedef map< string, **AL_SubjectTableInterface** * > **Subject_Tables**
  *A repository of **AL_SubjectTable** (p.53).*
- typedef **AL_Locket**< AL_SubjectRepository > **Guard**
  A repository guard.

6.11.2 Static Public Methods

- void **lock** ()
  Lock repository.
- void **unlock** ()
  Unlock repository.

6.11.3 Friends

- class **AL_SubjectRegistry**
  ***AL_SubjectRegistry** (p.50) can access repository.*

6.11.4 Member Typedef Documentation

6.11.4.1 typedef AL_Locket<AL_SubjectRepository> AL_SubjectRepository::Guard

A repository guard.

6.11.4.2 typedef map<string, AL_SubjectTableInterface *> AL_SubjectRepository::Subject_Tables

A repository of **AL_SubjectTable** (*p.53*).

6.11.5 Member Function Documentation

6.11.5.1 void AL_SubjectRepository::lock () `[static]`

Lock repository.

6.11.5.2 void AL_SubjectRepository::unlock () [static]

Unlock repository.

6.11.6 Friends And Related Function Documentation

6.11.6.1 friend class AL_SubjectRegistry [friend]

AL_SubjectRegistry (*p.50*) can access repository.

The documentation for this class was generated from the following file:

- **AL_SubjectRepository.h**

6.12 AL_SubjectTable Class Template Reference

AL_SubjectTableThe **AL_Subject** (*p.48*) table implementation.

`#include <AL_SubjectRepository.h>`

Inheritance diagram for AL_SubjectTable:

6.12.1 Public Types

- typedef map< string, **AL_Subject**< eventType, dataType >::Table > **Rep**
  *A repository table of **AL_Subject** (p.48).*

6.12.2 Public Methods

- **~AL_SubjectTable** ()
- **Rep** & **table** ()
  *The **AL_Subject** (p.48) table reference.*

6.12.3 Static Public Methods

- const char * **typeId** ()
  A unique type ID.

6.12.4 Dtailed Description

6.12.4.1 template<class eventType, class dataType> class AL_SubjectTable< eventType, dataType >

The **AL_Subject** (*p.48*) table implementation.

6.12.5 Member Typedef Documentation

6.12.5.1 template<class eventType, class dataType> typedef map<string,AL_Subject<eventType,dataType>::Table> AL_SubjectTable::Rep

A repository table of **AL_Subject** (*p.48*).

6.12.6 Constructor & Destructor Documentation

6.12.6.1 template<class eventType, class dataType> AL_SubjectTable< eventType, dataType >::~AL_SubjectTable () `[inline]`

6.12.7 Member Function Documentation

6.12.7.1 template<class eventType, class dataType> Rep& AL_SubjectTable< eventType, dataType >::table () [inline]

The **AL_Subject** (*p.48*) table reference.

6.12.7.2 template<class eventType, class dataType> const char* AL_SubjectTable< eventType, dataType >::typeId () [inline, static]

A unique type ID.

The documentation for this class was generated from the following file:
AL_SubjectRepository.h

7 Appendice B: Documentazione classi libreria XML-RPC

7.1 CommandHandler Class Reference

`#include <commandhandler.h>`

Collaboration diagram for CommandHandler:

7.1.1 Public Member Functions

- **CommandHandler** ()
- **~CommandHandler** ()
- int **open** (void *)
- int **handle_input** (ACE_HANDLE)
- ACE_HANDLE **get_handle** (void) const
- ACE_SOCK_Stream & **peer** (void) const

7.1.2 Static Public Member Functions

- void **SetCommandExecuter** (**ICommandExecuter** *ce)

7.1.3 Private Attributes

- ACE_SOCK_Stream **_peer**

7.1.4 Static Private Attributes

- **ICommandExecuter** * **_commandExecuter**

7.1.5 Detailed Description

handles the incoming command. It implement the "acceptor" and "reader" patterns. A thread (acceptor) wait the incoming connection and when a connection is "open" it wait the incoming stream command (reader).

7.1.6 Constructor & Destructor Documentation

7.1.6.1 CommandHandler::CommandHandler ()

7.1.6.2 CommandHandler::~CommandHandler ()

7.1.7 Member Function Documentation

7.1.7.1 ACE_HANDLE CommandHandler::get_handle (void) const

new method which returns the handle to the reactor when it asks for it.

7.1.7.2 int CommandHandler::handle_input (ACE_HANDLE)

Callback method invoked when a request arrives

7.1.7.3 int CommandHandler::open (void *)

7.1.7.4 ACE_SOCK_Stream & CommandHandler::peer (void) const

get a reference to the peer stream

7.1.7.5 void CommandHandler::SetCommandExecuter (ICommandExecuter * *ce*) [static]

7.1.8 Member Data Documentation

7.1.8.1 ICommandExecuter * CommandHandler::_commandExecuter [static, private]

the command executer object

7.1.8.2 ACE_SOCK_Stream CommandHandler::_peer [private]

Peer stream

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**commandhandler.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**commandhandler.cpp**

7.2 CommandSender Class Reference

`#include <commandsender.h>`

Collaboration diagram for CommandSender:

7.2.1 Public Member Functions

- **CommandSender** (char *hostname, int port)
- **~CommandSender** ()
- int **Connect** ()
- int **SendAsynch** (char *command)
  Uses a stream component to send data to the remote host.
- char * **SendSynch** (char *command)
  Uses a stream component to send data to the remote host.
- int **close** ()

7.2.2 Private Attributes

- ACE_SOCK_Stream **client_stream_**
- ACE_INET_Addr **remote_addr_**
- ACE_INET_Addr **local_addr_**
- ACE_SOCK_Connector **connector_**
- **XmlRpcUtil** * **_xmlRpcUtil**

7.2.3 Detailed Description

Send a command (via socket) to the other Remote Engine

7.2.4 Constructor & Destructor Documentation

7.2.4.1 CommandSender::CommandSender (char * *hostname*, int *port*)

7.2.4.2 CommandSender::~CommandSender ()

7.2.5 Member Function Documentation

7.2.5.1 int CommandSender::close ()

Close down the connection properly.

7.2.5.2 int CommandSender::Connect ()

Open the socket and connect it to the server

7.2.5.3 int CommandSender::SendAsynch (char * *command*)

Uses a stream component to send data to the remote host.
Send a command in a 'asyncronous' mode: it send the command without waiting the response

7.2.5.4 char * CommandSender::SendSynch (char * *command*)

Uses a stream component to send data to the remote host.
Send a command in a 'syncronous' mode: it send the command and wait the response.

7.2.6 Member Data Documentation

7.2.6.1 XmlRpcUtil* CommandSender::_xmlRpcUtil `[private]`

7.2.6.2 ACE_SOCK_Stream CommandSender::client_stream_ `[private]`

Socket stream

7.2.6.3 ACE_SOCK_Connector CommandSender::connector_ `[private]`

Connector

7.2.6.4 ACE_INET_Addr CommandSender::local_addr_ `[private]`

Remote Engine Address of the "client" (this process)

7.2.6.5 ACE_INET_Addr CommandSender::remote_addr_ `[private]`

Remote Engine Address of the "Server"

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**commandsender.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**commandsender.cpp**

7.3 ICommandExecuter Class Reference

`#include <icommandexecuter.h>`

7.3.1 Public Member Functions

- **ICommandExecuter** ()
- virtual **~ICommandExecuter** ()
- virtual char * **ExecuteCommand** (char *msg)=0
- virtual ACE_Reactor * **Reactor** ()=0
- virtual bool **IsSynch** ()=0

7.3.2 Detailed Description

7.3.2.1.1 Author:

assist user

7.3.3 Constructor & Destructor Documentation

7.3.3.1 ICommandExecuter::ICommandExecuter ()

7.3.3.2 ICommandExecuter::~ICommandExecuter () `[virtual]`

7.3.4 Member Function Documentation

7.3.4.1 virtual char* ICommandExecuter::ExecuteCommand (char * *msg*) `[pure virtual]`

7.3.4.2 virtual bool ICommandExecuter::IsSynch () `[pure virtual]`

Verify if the command executer is Synch or Asynch

7.3.4.3 virtual ACE_Reactor* ICommandExecuter::Reactor () `[pure virtual]`

Get the reactor instance

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**icommandexecuter.h**

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**icommandexecuter.cpp**

7.4 IConfigProvidePort Class Reference

`#include <iconfigprovideport.h>`

7.4.1 Public Member Functions

- virtual **~IConfigProvidePort** ()
- virtual long **connect** (char *ServerURI)=0
- virtual int **listen** ()=0
- virtual long **disconnect** ()=0
- virtual char * **getServerURI** ()=0

7.4.2 Detailed Description

An interface for the configuration of a Provide Port

7.4.2.1.1 Author:

Saverio Lombardo

7.4.3 Constructor & Destructor Documentation

7.4.3.1 virtual IConfigProvidePort::~IConfigProvidePort () [inline, virtual]

7.4.4 Member Function Documentation

7.4.4.1 virtual long IConfigProvidePort::connect (char * *ServerURI*) [pure virtual]

connect the port

7.4.4.1.1 Returns:

1 if all right or the error code if an error occour

7.4.4.2 virtual long IConfigProvidePort::disconnect () [pure virtual]

disconnect the port

7.4.4.2.1 Returns:

1 if all right or the error code if an error occour

7.4.4.3 virtual char* IConfigProvidePort::getServerURI () [pure virtual]

7.4.4.4 virtual int IConfigProvidePort::listen () [pure virtual]

The documentation for this class was generated from the following file:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**iconfigprovideport.h**

7.5 IConfigUsePort Class Reference

`#include <iconfiguseport.h>`

7.5.1 Public Member Functions

- virtual **~IConfigUsePort** ()
- virtual long **connect** (char *ServerURI)=0
- virtual long **disconnect** ()=0
- virtual char * **getServerURI** ()=0

7.5.2 Protected Attributes

- int **_serverPort**
- char * **_serverHost**
- char * **_serverURI**

7.5.3 Detailed Description

An interface for the configuration of a Use Port

7.5.3.1.1 Author:

Saverio Lombardo

7.5.4 Constructor & Destructor Documentation

7.5.4.1 virtual IConfigUsePort::~IConfigUsePort () [inline, virtual]

7.5.5 Member Function Documentation

7.5.5.1 virtual long IConfigUsePort::connect (char * *ServerURI*) [pure virtual]

connect the port

7.5.5.1.1 Returns:

1 if all right or the error code if an error occour

7.5.5.2 virtual long IConfigUsePort::disconnect () [pure virtual]

disconnect the port

7.5.5.2.1 Returns:

1 if all right or the error code if an error occour

7.5.5.3 virtual char* IConfigUsePort::getServerURI () [pure virtual]

7.5.6 Member Data Documentation

7.5.6.1 char* IConfigUsePort::_serverHost [protected]

7.5.6.2 int IConfigUsePort::_serverPort [protected]

7.5.6.3 char* IConfigUsePort::_serverURI [protected]

The documentation for this class was generated from the following file:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**iconfiguseport.h**

7.6 XMLParser Class Reference

`#include <xmlparser.h>`

7.6.1 Public Member Functions

- **XMLParser** ()
- **~XMLParser** ()
- DOMNode * **getNode** (char *ElementName, long i=0)
- DOMNode * **getNode** (DOMNode *parent, char *ElementName, long i=0)
- const char * **getElement** (char *, long)
- const char * **getChildValue** (DOMNode *parent, long i=0)
- const char * **getChildValue** (DOMNode *parent, char *nodeName, long i=0)
- const char * **getChildName** (char *parentName, int i=0)
- const char * **getValue** (char *nodeName, long i=0)
- char * **getFragment** (char *nodeName, long i=0)
- char * **getFragment** (DOMNode *parent, char *nodeName, long i=0)
- int **setStream** (char *)
- int **OpenDocument** (char *URL, bool validate=false)
- long **getLengthElement** (char *ElementName)
- long **getLengthElement** (DOMNode *node, char *ElementName)
- int **init** ()
- int **freeMemory** ()
- int **reset** ()

7.6.2 Private Member Functions

- char * **print_to_string** (DOMNode *node)

7.6.3 Private Attributes

- XercesDOMParser * **parser**
- ErrorHandler * **errHandler**
- char * **input_string**
- MemBufInputSource * **input_stream**
- DOMDocument * **document**

7.6.4 Constructor & Destructor Documentation

7.6.4.1 XMLParser::XMLParser ()

7.6.4.2 XMLParser::~XMLParser ()

7.6.5 Member Function Documentation

7.6.5.1 int XMLParser::freeMemory ()

free the memory of the parser. It is necessary for the re-use of the parser

7.6.5.2 const char * XMLParser::getChildName (char * *parentName*, int *i* = 0)

7.6.5.3 const char * XMLParser::getChildValue (DOMNode * *parent*, char * *nodeName*, long *i* = 0)

7.6.5.4 const char * XMLParser::getChildValue (DOMNode * *parent*, long *i* = 0)

7.6.5.5 const char * XMLParser::getElement (char *, long)

7.6.5.6 char * XMLParser::getFragment (DOMNode * *parent*, char * *nodeName*, long *i* = 0)

7.6.5.7 char * XMLParser::getFragment (char * *nodeName*, long *i* = 0)

7.6.5.8 long XMLParser::getLengthElement (DOMNode * *node*, char * *ElementName*)

7.6.5.9 long XMLParser::getLengthElement (char * *ElementName*)

7.6.5.10 DOMNode * XMLParser::getNode (DOMNode * *parent*, char * *ElementName*, long *i* = 0)

7.6.5.11 DOMNode * XMLParser::getNode (char * *ElementName*, long *i* = 0)

7.6.5.12 const char * XMLParser::getValue (char * *nodeName*, long *i* = 0)

7.6.5.13 int XMLParser::init ()

Init the parser: you should call it before using the Parser

7.6.5.14 int XMLParser::OpenDocument (char * *URL*, bool *validate* = `false`)

7.6.5.15 char * XMLParser::print_to_string (DOMNode * *node*) `[private]`

create a string that contains the XML tree

7.6.5.16 int XMLParser::reset ()

reset the Writer for a new operation

7.6.5.17 int XMLParser::setStream (char * *is*)

set the input stream (char *) and create the DOM tree

7.6.6 Member Data Documentation

7.6.6.1 DOMDocument* XMLParser::document `[private]`

7.6.6.2 ErrorHandler* XMLParser::errHandler `[private]`

7.6.6.3 MemBufInputSource* XMLParser::input_stream `[private]`

7.6.6.4 char* XMLParser::input_string `[private]`

7.6.6.5 XercesDOMParser* XMLParser::parser `[private]`

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlparser.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlparser.cpp**

7.7 XmlRpcData Class Reference

`#include <xmlrpcdata.h>`

7.7.1 Public Types

- typedef vector< char * > **BinaryData**
- typedef vector< **XmlRpcData** * > **ValueArray**
- typedef map< string, **XmlRpcData** * > **ValueStruct**
- enum **Type** { **TypeInvalid**, **TypeBoolean**, **TypeInt**, **TypeDouble**, **TypeString**, **TypeDateTime**, **TypeBase64**, **TypeArray**, **TypeStruct** }

7.7.2 Public Member Functions

- **~XmlRpcData** ()
- **XmlRpcData** ()
- **XmlRpcData** (bool value)
- **XmlRpcData** (int value)
- **XmlRpcData** (double value)
- **XmlRpcData** (char *value)
- **XmlRpcData** (struct tm *value)
- **XmlRpcData** & **operator=** (**XmlRpcData** const &rhs)
- **XmlRpcData** & **operator=** (int rhs)
- **XmlRpcData** & **operator=** (double rhs)
- **XmlRpcData** & **operator=** (char *rhs)
- **XmlRpcData** & **operator[]** (int i)
- **XmlRpcData** & **operator[]** (char *name)
- bool **getBool** ()
- int **getInt** ()
- char * **getString** ()
- tm **getTime** ()
- double **getDouble** ()
- void **Clean** ()
- bool **Valid** ()
- **Type GetType** ()
- void **SetType** (**Type** t)
- bool **FromXml** (char *valueXml)
- char * **ToXml** ()
- unsigned int **Size** ()
- bool **AddToStruct** (char *memberName, **XmlRpcData** *member)
- bool **AddToList** (**XmlRpcData** *member)

7.7.3 Protected Attributes

- **Type _type**
- bool **_boolValue**
- int **_intValue**
- double **_doubleValue**
- tm * **_timeValue**
- char * **_stringValue**
- **BinaryData** * **_binaryValue**
- **ValueArray** * **_arrayValue**

- **ValueStruct * _structValue**

7.7.4 Private Member Functions

- bool **BoolFromXml** (char *valueXml)
- bool **IntFromXml** (char *valueXml)
- bool **DoubleFromXml** (char *valueXml)
- bool **StringFromXml** (char *valueXml)
- bool **TimeFromXml** (char *valueXml)
- bool **BinaryFromXml** (char *valueXml)
- bool **ArrayFromXml** (char *valueXml)
- bool **StructFromXml** (char *valueXml)
- char * **BoolToXml** ()
- char * **IntToXml** ()
- char * **DoubleToXml** ()
- char * **StringToXml** ()
- char * **TimeToXml** ()
- char * **BinaryToXml** ()
- char * **ArrayToXml** ()
- char * **StructToXml** ()

7.7.5 Static Private Attributes

- const char **VALUE_TAG** [] = "<value>"
- const char **VALUE_ETAG** [] = "</value>"
- const char **BOOLEAN_TAG** [] = "<boolean>"
- const char **BOOLEAN_ETAG** [] = "</boolean>"
- const char **DOUBLE_TAG** [] = "<double>"
- const char **DOUBLE_ETAG** [] = "</double>"
- const char **INT_TAG** [] = "<int>"
- const char **INT_ETAG** [] = "</int>"
- const char **I4_TAG** [] = "<i4>"
- const char **I4_ETAG** [] = "</i4>"
- const char **STRING_TAG** [] = "<string>"
- const char **STRING_ETAG** [] = "</string>"
- const char **DATETIME_TAG** [] = "<dateTime.iso8601>"
- const char **DATETIME_ETAG** [] = "</dateTime.iso8601>"
- const char **BASE64_TAG** [] = "<base64>"
- const char **BASE64_ETAG** [] = "</base64>"
- const char **ARRAY_TAG** [] = "<array>"
- const char **DATA_TAG** [] = "<data>"
- const char **DATA_ETAG** [] = "</data>"
- const char **ARRAY_ETAG** [] = "</array>"
- const char **STRUCT_TAG** [] = "<struct>"
- const char **MEMBER_TAG** [] = "<member>"
- const char **NAME_TAG** [] = "<name>"
- const char **NAME_ETAG** [] = "</name>"
- const char **MEMBER_ETAG** [] = "</member>"
- const char **STRUCT_ETAG** [] = "</struct>"
- const char **VALUE** [] = "value"
- const char **BOOLEAN** [] = "boolean"
- const char **DOUBLE** [] = "double"
- const char **INT** [] = "int"

- const char **I4** [] = "i4"
- const char **STRING** [] = "string"
- const char **DATETIME** [] = "dateTime.iso8601"
- const char **BASE64** [] = "base64"
- const char **DATA** [] = "data"
- const char **ARRAY** [] = "array"
- const char **NAME** [] = "name"
- const char **MEMBER** [] = "member"
- const char **STRUCT** [] = "struct"

7.7.6 Detailed Description

7.7.6.1.1 Author:

Saverio Lombardo

7.7.7 Member Typedef Documentation

7.7.7.1 typedef vector<char *> XmlRpcData::BinaryData

7.7.7.2 typedef vector<XmlRpcData *> XmlRpcData::ValueArray

7.7.7.3 typedef map<string, XmlRpcData *> XmlRpcData::ValueStruct

---

7.7.8 Member Enumeration Documentation

7.7.8.1 enum XmlRpcData::Type

Enumeration values:

TypeInvalid
TypeBoolean
TypeInt
TypeDouble
TypeString
TypeDateTime
TypeBase64
TypeArray
TypeStruct

7.7.9 Constructor & Destructor Documentation

7.7.9.1 XmlRpcData::~XmlRpcData ()

7.7.9.2 XmlRpcData::XmlRpcData () `[inline]`

7.7.9.3 XmlRpcData::XmlRpcData (bool *value*) `[inline]`

7.7.9.4 XmlRpcData::XmlRpcData (int *value*) `[inline]`

7.7.9.5 XmlRpcData::XmlRpcData (double *value*) `[inline]`

7.7.9.6 XmlRpcData::XmlRpcData (char * *value*) `[inline]`

7.7.9.7 XmlRpcData::XmlRpcData (struct tm * *value*) `[inline]`

7.7.10 Member Function Documentation

7.7.10.1 bool XmlRpcData::AddToList (XmlRpcData * *member*)

7.7.10.2 bool XmlRpcData::AddToStruct (char * *memberName*, XmlRpcData * *member*)

7.7.10.3 bool XmlRpcData::ArrayFromXml (char * *valueXml*) [private]

7.7.10.4 char * XmlRpcData::ArrayToXml () [private]

7.7.10.5 bool XmlRpcData::BinaryFromXml (char * *valueXml*) [private]

7.7.10.6 char * XmlRpcData::BinaryToXml () [private]

7.7.10.7 bool XmlRpcData::BoolFromXml (char * *valueXml*) [private]

7.7.10.8 char * XmlRpcData::BoolToXml () [private]

7.7.10.9 void XmlRpcData::Clean ()

Erase all the content of the class

7.7.10.10 bool XmlRpcData::DoubleFromXml (char * *valueXml*) [private]

7.7.10.11 char * XmlRpcData::DoubleToXml () [private]

7.7.10.12 bool XmlRpcData::FromXml (char * *valueXml*)

Decode xml. Destroys any existing value.

7.7.10.13 bool XmlRpcData::getBool () [inline]

7.7.10.14 double XmlRpcData::getDouble () [inline]

7.7.10.15 int XmlRpcData::getInt () [inline]

7.7.10.16 char* XmlRpcData::getString () [inline]

7.7.10.17 struct tm XmlRpcData::getTime () [inline]

7.7.10.18 Type XmlRpcData::GetType () [inline]

Return the type of the value stored.

7.7.10.19 bool XmlRpcData::IntFromXml (char * *valueXml*) [private]

7.7.10.20 char * XmlRpcData::IntToXml () [private]

7.7.10.21 XmlRpcData& XmlRpcData::operator= (char * *rhs*) [inline]

7.7.10.22 XmlRpcData& XmlRpcData::operator= (double *rhs*) [inline]

7.7.10.23 XmlRpcData& XmlRpcData::operator= (int *rhs*) [inline]

7.7.10.24 XmlRpcData & XmlRpcData::operator= (XmlRpcData const & *rhs*)

7.7.10.25 XmlRpcData& XmlRpcData::operator[] (char * *name*) [inline]

7.7.10.26 XmlRpcData& XmlRpcData::operator[] (int *i*) [inline]

7.7.10.27 void XmlRpcData::SetType (Type *t*) [inline]

7.7.10.28 unsigned int XmlRpcData::Size ()

Return the size for string, base64, array, and struct values

7.7.10.29 bool XmlRpcData::StringFromXml (char * *valueXml*) [private]

7.7.10.30 char * XmlRpcData::StringToXml () [private]

7.7.10.31 bool XmlRpcData::StructFromXml (char * *valueXml*) [private]

7.7.10.32 char * XmlRpcData::StructToXml () [private]

7.7.10.33 bool XmlRpcData::TimeFromXml (char * *valueXml*) [private]

7.7.10.34 char * XmlRpcData::TimeToXml () [private]

7.7.10.35 char * XmlRpcData::ToXml ()

Encode the Value in xml

7.7.10.36 bool XmlRpcData::Valid () [inline]

Return true if the value has been set to something.

7.7.11 Member Data Documentation

7.7.11.1 ValueArray* XmlRpcData::_arrayValue [protected]

7.7.11.2 BinaryData* XmlRpcData::_binaryValue [protected]

7.7.11.3 bool XmlRpcData::_boolValue [protected]

7.7.11.4 double XmlRpcData::_doubleValue [protected]

7.7.11.5 int XmlRpcData::_intValue [protected]

7.7.11.6 char* XmlRpcData::_stringValue [protected]

7.7.11.7 ValueStruct* XmlRpcData::_structValue [protected]

7.7.11.8 struct tm* XmlRpcData::_timeValue [protected]

7.7.11.9 Type XmlRpcData::_type [protected]

7.7.11.10 const char XmlRpcData::ARRAY = "array" [static, private]

7.7.11.11 const char XmlRpcData::ARRAY_ETAG = "</array>" [static, private]

7.7.11.12 const char XmlRpcData::ARRAY_TAG = "<array>" [static, private]

7.7.11.13 const char XmlRpcData::BASE64 = "base64" [static, private]

7.7.11.14 const char XmlRpcData::BASE64_ETAG = "</base64>" [static, private]

7.7.11.15 const char XmlRpcData::BASE64_TAG = "<base64>" [static, private]

7.7.11.16 const char XmlRpcData::BOOLEAN = "boolean" [static, private]

7.7.11.17 const char XmlRpcData::BOOLEAN_ETAG = "</boolean>" [static, private]

7.7.11.18 const char XmlRpcData::BOOLEAN_TAG = "<boolean>" [static, private]

7.7.11.19 const char XmlRpcData::DATA = "data" [static, private]

7.7.11.20 const char XmlRpcData::DATA_ETAG = "</data>" [static, private]

7.7.11.21 const char XmlRpcData::DATA_TAG = "<data>" [static, private]

7.7.11.22 const char XmlRpcData::DATETIME = "dateTime.iso8601" [static, private]

7.7.11.23 const char XmlRpcData::DATETIME_ETAG = "</dateTime.iso8601>" [static, private]

7.7.11.24 const char XmlRpcData::DATETIME_TAG = "<dateTime.iso8601>" [static, private]

7.7.11.25 const char XmlRpcData::DOUBLE = "double" [static, private]

7.7.11.26 const char XmlRpcData::DOUBLE_ETAG = "</double>" [static, private]

7.7.11.27 const char XmlRpcData::DOUBLE_TAG = "<double>" [static, private]

7.7.11.28 const char XmlRpcData::I4 = "i4" [static, private]

7.7.11.29 const char XmlRpcData::I4_ETAG = "</i4>" [static, private]

7.7.11.30 const char XmlRpcData::I4_TAG = "<i4>" [static, private]

7.7.11.31 const char XmlRpcData::INT = "int" [static, private]

7.7.11.32 const char XmlRpcData::INT_ETAG = "</int>" [static, private]

7.7.11.33 const char XmlRpcData::INT_TAG = "<int>" [static, private]

7.7.11.34 const char XmlRpcData::MEMBER = "member" [static, private]

7.7.11.35 const char XmlRpcData::MEMBER_ETAG = "</member>" [static, private]

7.7.11.36 const char XmlRpcData::MEMBER_TAG = "<member>" [static, private]

7.7.11.37 const char XmlRpcData::NAME = "name" [static, private]

7.7.11.38 const char XmlRpcData::NAME_ETAG = "</name>" [static, private]

7.7.11.39 const char XmlRpcData::NAME_TAG = "<name>" [static, private]

7.7.11.40 const char XmlRpcData::STRING = "string" [static, private]

7.7.11.41 const char XmlRpcData::STRING_ETAG = "</string>" [static, private]

7.7.11.42 const char XmlRpcData::STRING_TAG = "<string>" [static, private]

7.7.11.43 const char XmlRpcData::STRUCT = "struct" [static, private]

7.7.11.44 const char XmlRpcData::STRUCT_ETAG = "</struct>" [static, private]

7.7.11.45 const char XmlRpcData::STRUCT_TAG = "<struct>" [static, private]

7.7.11.46 const char XmlRpcData::VALUE = "value" [static, private]

7.7.11.47 const char XmlRpcData::VALUE_ETAG = "</value>" [static, private]

7.7.11.48 const char XmlRpcData::VALUE_TAG = "<value>" [static, private]

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlrpcdata.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlrpcdata.cpp**

7.8 XmlRpcUtil Class Reference

`#include <xmlrpcutil.h>`

Collaboration diagram for XmlRpcUtil:

7.8.1 Public Member Functions

- **XmlRpcUtil** ()
- **~XmlRpcUtil** ()
- char * **GenerateRequest** (char *methodName, vector< **XmlRpcData** * > params)
- char * **ParseRequest** (vector< **XmlRpcData** * > *params, char *request)
- char * **GenerateRequest** (char *methodName, **XmlRpcData** *param)
- char * **GenerateResponse** (**XmlRpcData** *result)
- char * **GenerateFaultResponse** (**XmlRpcData** *result)
- bool **ParseResponse** (**XmlRpcData** *result, char *response)
- bool **ParseFaultResponse** (**XmlRpcData** *result, char *response)

7.8.2 Static Public Attributes

- const char **METHOD_NAME** [] = "methodName"
- const char **METHOD_CALL** [] = "methodCall"
- const char **PARAMS** [] = "params"
- const char **PARAM** [] = "param"
- const char **VALUE** [] = "value"
- const char **METHOD_RESPONSE** [] = "methodReponse"
- const char **FAULT** [] = "fault"

7.8.3 Private Attributes

- **XMLParser** * **_xmlParser**
- **XMLWriter** * **_xmlWriter**
- DOMDocument * **_xmlDoc**

7.8.4 Detailed Description

Utilities for XML parsing, encoding, and decoding and message handlers.

7.8.4.1.1 Author:

Saverio Lombardo

7.8.5 Constructor & Destructor Documentation

7.8.5.1 XmlRpcUtil::XmlRpcUtil ()

7.8.5.2 XmlRpcUtil::~XmlRpcUtil ()

7.8.6 Member Function Documentation

7.8.6.1 char * XmlRpcUtil::GenerateFaultResponse (XmlRpcData * *result*)

7.8.6.2 char * XmlRpcUtil::GenerateRequest (char * *methodName*, XmlRpcData * *param*)

7.8.6.3 char * XmlRpcUtil::GenerateRequest (char * *methodName*, vector< XmlRpcData * > *params*)

Generate an xml-rpc request

7.8.6.4 char * XmlRpcUtil::GenerateResponse (XmlRpcData * *result*)

7.8.6.5 bool XmlRpcUtil::ParseFaultResponse (XmlRpcData * *result*, char * *response*)

7.8.6.6 char * XmlRpcUtil::ParseRequest (vector< XmlRpcData * > * *params*, char * *request*)

7.8.6.7 bool XmlRpcUtil::ParseResponse (XmlRpcData * *result*, char * *response*)

7.8.7 Member Data Documentation

7.8.7.1 DOMDocument* XmlRpcUtil::_xmlDoc `[private]`

XML document

7.8.7.2 XMLParser* XmlRpcUtil::_xmlParser [private]

XML Parser

7.8.7.3 XMLWriter* XmlRpcUtil::_xmlWriter [private]

XML document writer

7.8.7.4 const char XmlRpcUtil::FAULT = "fault" [static]

7.8.7.5 const char XmlRpcUtil::METHOD_CALL = "methodCall" [static]

7.8.7.6 const char XmlRpcUtil::METHOD_NAME = "methodName" [static]

7.8.7.7 const char XmlRpcUtil::METHOD_RESPONSE = "methodReponse" [static]

7.8.7.8 const char XmlRpcUtil::PARAM = "param" [static]

7.8.7.9 const char XmlRpcUtil::PARAMS = "params" [static]

7.8.7.10 const char XmlRpcUtil::VALUE = "value" [static]

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlrpcutil.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlrpcutil.cpp**

7.9 XMLWriter Class Reference

`#include <xmlwriter.h>`

7.9.1 Public Member Functions

- **XMLWriter** ()
- **~XMLWriter** ()
- int **init** ()
- DOMDocument * **create_document** (char *)
- int **reset** ()
- char * **print_to_string** ()
- DOMElement * **add_text_node** (DOMElement ***Node**, char *name, char *val)
- DOMElement * **add_node** (DOMElement ***Node**, char *name)
- void **add_fragment** (DOMElement *parent, char *name, char *val)

7.9.2 Private Attributes

- DOMImplementation * **impl**
- DOMDocument * **doc**
- DOMWriter * **writer**
- MemBufFormatTarget * **formTarget**

7.9.3 Detailed Description

7.9.3.1.1 Author:

assist user

7.9.4 Constructor & Destructor Documentation

7.9.4.1 XMLWriter::XMLWriter ()

7.9.4.2 XMLWriter::~XMLWriter ()

7.9.5 Member Function Documentation

7.9.5.1 void XMLWriter::add_fragment (DOMElement * *parent*, char * *name*, char * *val*)

7.9.5.2 DOMElement * XMLWriter::add_node (DOMElement * *Node*, char * *name*)

7.9.5.3 DOMElement * XMLWriter::add_text_node (DOMElement * *Node*, char * *name*, char * *val*)

7.9.5.4 DOMDocument * XMLWriter::create_document (char *)

7.9.5.5 int XMLWriter::init ()

7.9.5.6 char * XMLWriter::print_to_string ()

create a string that contains the XML tree

7.9.5.7 int XMLWriter::reset ()

reset the Writer for a new operation

7.9.6 Member Data Documentation

7.9.6.1 DOMDocument* XMLWriter::doc `[private]`

7.9.6.2 MemBufFormatTarget* XMLWriter::formTarget `[private]`

7.9.6.3 DOMImplementation* XMLWriter::impl `[private]`

7.9.6.4 DOMWriter* XMLWriter::writer `[private]`

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlwriter.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xmlwriter.cpp**

7.10 XStr Class Reference

`#include <xstr.h>`

7.10.1 Public Member Functions

- **XStr** (const char *const)
- **~XStr** ()
- const XMLCh * **unicodeForm** () const

7.10.2 Private Attributes

- XMLCh * **fUnicodeForm**

7.10.3 Constructor & Destructor Documentation

7.10.3.1 XStr::XStr (const char * *const*)

7.10.3.2 XStr::~XStr ()

7.10.4 Member Function Documentation

7.10.4.1 const XMLCh * XStr::unicodeForm () const

7.10.5 Member Data Documentation

7.10.5.1 XMLCh* XStr::fUnicodeForm `[private]`

This is the Unicode XMLCh format of the string.

The documentation for this class was generated from the following files:

- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xstr.h**
- D:/ICAR/VPG/backup/vpg_05_01_15_version_1_2/vpg/Utilities/**xstr.c**