Università degli Studi di Catania

Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica

Wrapper C++ per Python cross-compilato per architetture ARM

Giuseppe Astuti

Relatore
Chiar.mo Prof. S. Riccobene

Correlatore
Dott. G. Patanè

Anno Accademico 2015/16

Indice dei contenuti

Introduzione

L'elaborato si propone di presentare una soluzione al problema, promosso da ParkSmart srl[1], relativo alla creazione di un wrapper per interfacciare librerie C++ a Python.
ParkSmart è una startup che opera nel campo delle Smart Cities, proponendo un progetto innovativo il cui scopo è monitorare gli stalli di parcheggio bordo strada e comunicare ai cittadini, mediante un'apposita applicazione per smartphone, dove trovare parcheggio, migliorando la qualità della vita in città.

[1] www.parksmart.it

1
Problema

ParkSmart ha sviluppato una libreria C++ di computer vision che sfrutta le accelerazioni fornite da GPU Cuda. Lo sviluppo in C++ però non si presta bene alla prototipazione veloce, mentre in Python con poche righe di codice si riescono a prototipare rapidamente soluzioni a problemi diversi, consentendo di testare diverse soluzioni fino ad individuare quella che meglio risolve il problema che si vuole affrontare. A tal proposito si è deciso di rendere disponibile la libreria C++ da Python, andando a creare un wrapper di interfacciamento Python/C++.

Diversi sono i vantaggi che questa soluzione porta, a partire dall'efficienza computazionale che si riesce ad offrire dato che C++, grazie alla tipizzazione e al basso livello di astrazione, è uno dei linguaggi di programmazione più potenti ed efficienti. C++ però non è alla portata di tutti, infatti creare procedure efficienti che sfruttano librerie per computare su GPU molto spesso è una via insidiosa, via che a volte solo programmatori esperti possono seguire.
In ambito aziendale scrivere procedure in C++ è un limite, dato che, saranno alla portata di pochi. Offrire una soluzione come quella che l'elaborato propone fornisce la possibilità di *mettere a lavoro* anche sviluppatori con meno esperienza.

2
Obiettivi

Gli obiettivi, che stanno anche alla base del design pattern Adapter, sono semplici ma importanti. Obiettivo primario è quello di fornire un'interfaccia semplificata ad alto livello in modo da invocare in maniera efficiente procedure scritte in C++ che potrebbero essere, invece, operazioni onerose computazionalmente se scritte in Python. Scrivere le procedure di cui ci stiamo occupando solamente in C++ significherebbe altissima efficienza in termini computazionali ma altresì difficoltà al momento dell'invocazione; scrivere in Python sarebbe invece più semplice ma particolarmente oneroso in termini di prestazioni. Scrivere una soluzione ibrida, creando un wrapper che adatta Python a C++ manterrebbe alto il livello di efficienza computazionale con particolare semplicità di utilizzo senza dover compilare al momento dell'invocazione.

2.1 Implementazione del design pattern Adapter: Wrapper

L'idea è derivata dal famoso design pattern "Adapter", in genere utilizzato nella programmazione orientata agli oggetti, chiamato anche wrapper (ovvero involucro)

per lo schema di funzionamento.

Il fine di Adapter è quello di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il design pattern Adapter "converte l'interfaccia di una classe in un'altra interfaccia che il cliente si aspetta. Adapter permette a delle classi di lavorare insieme quando normalmente non potrebbero a causa di incompatibilità di interfacce.

Motivazione

Adapter è un pattern strutturale basato su classi o su oggetti in quanto è possibile ottenere entrambe le rappresentazioni. Viene utilizzato quando si intende utilizzare un componente software ma occorre adattare la sua interfaccia per motivi di integrazione con l'applicazione esistente.

Questo comporta la definizione di una nuova interfaccia che deve essere compatibile con quella esistente in modo tale da consentire la comunicazione con l'interfaccia da "adattare". In tale contesto possono essere anche effettuate delle trasformazioni di dati per cui l'Adapter si occuperà di interfacciarsi con il nuovo sistema e fornisce anche le regole di mapping dei dati. Come abbiamo accennato, tale pattern può essere basato sia su classi che su oggetti pertanto l'instanza della classe da adattare può derivare da ereditarietà oppure da associazione.

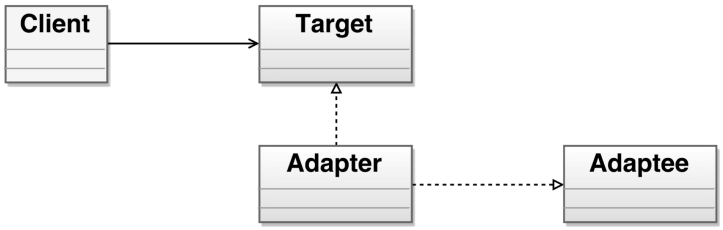
Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

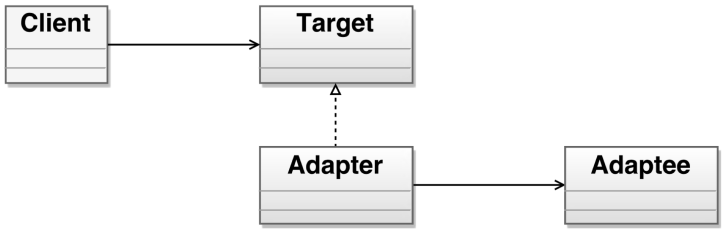
1. **Client**: colui che effettua l'invocazione all'operazione di interesse.
2. **Target**: definisce l'interfaccia specifica del dominio applicativo utilizzata dal Client.
3. **Adaptee**: definisce l'interfaccia di un diverso dominio applicativo da dover adattare per l'invocazione da parte del Client.
4. **Adapter**: definisce l'interfaccia compatibile con il Target che maschera l'invocazione dell'Adaptee.

Il pattern può essere basato su Classi o su Oggetti, in base a questo è possibile schematizzare in UML la relazione esistente tra l'adattore e l'adattato (Adapter e Adaptee).

- Sotto forma di ereditarietà come nel caso seguente:



- O sotto forma di associazione:



Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Class Adapter: prevede un rapporto di ereditarietà tra Adapter e Adaptee, in cui Adapter specializza Adaptee, pertanto non è possibile creare un Adapter che specializzi più Adaptee. Se esiste una gerarchia di Adaptee occorre creare una gerarchia di Adapter.

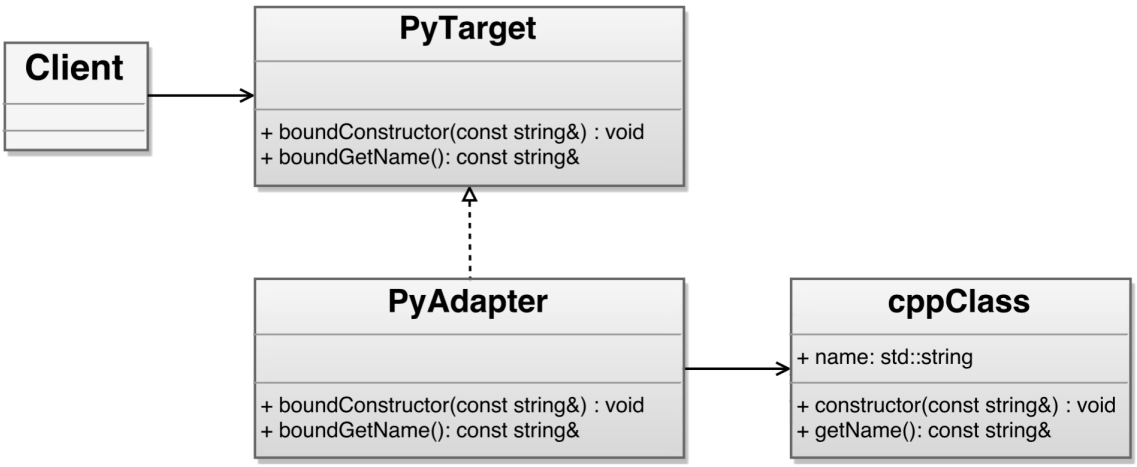
2. Object Adapter: prevede un rapporto di associazione tra Adapter e Adaptee, in cui Adapter instanzia Adaptee, pertanto è anche possibile avere un Adapter associato con più Adaptee.

Implementazione

Come esempio possiamo pensare ad una semplice classe scritta in C++ il cui costruttore viene chiamato con una stringa in parametro ed un metodo pubblico `std::string getName() const` che ritorna la stringa iniziale. L'esempio, come vedremo successivamente, si avvicina all'header del Detector di ParkSmart scritto in C++.

Essendo un binding da Python ad una libreria C++ è conveniente usare la soluzione **Object Adapter**, in modo tale che la classe Python che fa da Adapter, determini i metodi dell'Adaptee, scritto in C++, che vuole esporre avendo il pieno controllo sulle chiamate al codice C++.

Di seguito la rappresentazione UML del nostro esempio usando Object Adapter.



3
Requisiti

3.1 Cross-compilazione

Non è sempre possibile scrivere e compilare un applicazione sulla stessa piattaforma. Per alcuni sistemi embedded, il ridotto spazio di memoria, a volte meno di 256MB o anche meno di 64MB sia per RAM che per storage, rappresenta un limite. Un normale compilatore C, con i tool ad esso associati, e le Librerie C richieste non potranno mai entrare in ambienti di queste dimensioni, ne tanto meno essere eseguiti.

Se si parla poi di sviluppo la situazione tende ad aggravarsi. Lussi come un editor di testo, o di un vero e proprio ambiente di sviluppo non sono disponibili, partendo dal presupposto che può accadere che il dispositivo abbia un display minuscolo, o che non supporti tastiere. Molti sistemi embedded non hanno nemmeno l'accesso alla rete, quindi non si possono controllare o programmare da remoto.

I cross-compilatori permettono di sviluppare su una piattaforma (host) su cui si effettua il "build" per un sistema alterativo (target). La macchina target non ha bisogno di essere fisicamente accessibile, tutto ciò di cui si ha bisogno è sapere come scrivere codice per la macchina target. I cross-compilatori possono essere utili anche in altre situazioni. Si ponga per assurdo che una macchina target non abbia un

compilatore C installato e non si abbia il modo di ottenere un binario precompilato. Con il codice sorgente per GNU Compiler Collection (GCC), una Libreria C (newlib) e le binutils in un computer con compilatore C sarebbe possibile creare da zero un pacchetto completo da copiare sulla macchina target ed eseguirlo.

I cross-compilatori possono essere utili anche in presenza di macchine con scarse prestazioni o molto lente. Con una cross-compilazione su una macchina host si potrebbero evitare ore o addirittura giorni di compilazione, risolvendo in pochi minuti in problema.

3.1.1 Funzionamento

In questo elaborato verrà discusso il funzionamento di GCC, ma i principi di base valgono per tutti i compilatori. Differenti componenti lavorano insieme, con il fine ultimo di generare un binarycode che la CPU esegue. Quando il binarycode sarà "assemblato", allora la cross-compilazione sarà avvenuta con successo.

Le principali componenti di un compilatore sono:

- Parser: Il Parser converte il codice sorgente in linguaggio assembly. Data la conversione (ad esempio da C ad assembly), il parser deve conoscere il linguaggio assembly target.

- Assembler: L'assembler converte il codice assembly in binarycode che la CPU esegue.

- Linker: Il linker combina i singoli file object che l'assembler genera in applicazioni eseguibili. Diverse combinazioni di sistemi operativi e CPU usano differenti meccanismi di incapsulamento e dunque il linker ha bisogno di sapere quale sarà il formato target su cui lavorare.

- Librerie C Standard: Il core delle funzioni C (per esempio, printf) sono scritte in una libreria C fondamentale. Questa libreria è usata combinandola con il linker e il codice sorgente per creare un eseguibile finale, ipotizzando che le funzioni di questa libreria vengano usate nell'applicazione.

In un compilatore C standard, ognuna di queste componenti è pensata per produrre il corrispondente codice assembly, binarycode e l'eseguibile per la determinata macchina host. In un cross-compilatore invece, anche se l'applicazione è scritta per essere eseguita sull'host, il codice assembly, il linker e le librerie C sono pensate per essere eseguite sulla macchina target con una determinata architettura. Per esempio è possibile generare con una macchina Intel con Linux, un'applicazione per una macchina SPARC con Solaris, o come nel progetto presentato da questo elaborato, per ARM con Linux.

Generare un cross-compilatore vuol dire quindi costruire una versione alternativa rispetto alla suite standard di C; compilando con GCC e diversi tool associati è possibile costruire un cross-compilatore ad hoc.

3.1.2 Creazione di un cross-compilatore

Le utilities GNU (chiamate GCC), che includono il compilatore C, le binutils e le librerie C, hanno dei vantaggi rispetto ad altri compilatori. Oltre al fatto che sono open source e semplici da compilare, hanno il supporto della maggior parte delle CPU e per differenti piattaforme. Per creare un cross-compilatore si ha bisogno di compilare tre componenti dalla suite GNU:

- binutils: Il pacchetto binutils racchiude le utility base come l'Assembler, il Linker, e tools ad esso associati come Size e Strip. Binutils contiene le componenti fondamentali per compilare un'applicazione e per manipolare l'esecuzione finale. Ad esempio, Strip rimuove le Symbol table, le informazioni di Debug, e altre informazioni "superflue" (per i binari da produzione ma utili per le fasi di test e debug) da un file object o da un'applicazione, ma per fare questo le binutils hanno bisogno di sapere quale sarà il formato target per non rimuovere l'informazione errata.

- gcc: Il principale componente della compilazione è gcc che comprende il preprocessore C (cpp) e il traduttore, che converte il codice C nell'assembly che il processore si aspetta. Chiamando cpp, il traduttore, l'assembler e il

linker in maniera accurata, si permette a gcc di figurare come interfaccia tra processi.

- newlib/glibc: Questa è la libreria C standard. Newlib è stata sviluppata grazie a Redhat e viene utilizzata spesso nella cross-compilazione con target embedded.

A volte, se non si usano sistemi Linux, è necessario dover copiare i file header nel sistema operativo target, per poter accedere a tutte le funzioni di sistema e alle chiamate di sistema che servono ad una determinata applicazione. Il set di GNU prevede l'opzione di debug sulla macchina host.

3.1.3 Processo di cross-compilazione

Si presenta una piccola panoramica sui passaggi del processo di cross-compilazione.

1. Il compilatore C `gcc` chiama il preprocessore C `cpp` su un sorgente in input, di solito un file "c", producendo un file ".i" che continua ad essere codice C ma privo dalle direttive del preprocessore. Questo processo è invisibile all'utente.

2. `gcc` richiama il `cc1` con l'output del `cpp` producendo un file ".s". Si tratta di un file sorgente assembler corrispondente al file ".c" originario.

3. `gcc` richiama l'assembler con l'output di `cc1` creando un file ".o", il risultato è un file "object" che corrisponde al file assembler in input.

4. `gcc` chiama il linker `ld` con l'output di `as` e altri file (a volte chiamati "crtstuff"), per creare un file eseguibile.

Insieme al file eseguibile vengono creati archivi con il programma binutils derivato dal file object. Le librerie condivise invece, vengono create chiamando insieme al comando `gcc` il parametro `-shared`, che richiama il comando `ld` con varie opzioni.

3.2 Perché usare C++

C++ è utilizzato da centinaia di migliaia di programmatori praticamente in qualsiasi dominio applicativo. Ciò è agevolato dall'esistenza di circa una dozzina di

implementazioni diverse, centinaia di librerie, centinaia di libri, diverse riviste tecniche, molte conferenze e tantissimi consulenti. Il suo insegnamento a qualsiasi livello è facilmente disponibile a chiunque. Come il nome può suggerire il C++ è un evoluzione di C: la differenza riguarda soprattutto l'introduzione del paradigma di programmazione ad oggetti, ragion per cui il C++ è detto anche "C con classi".

Tra i vantaggi che C++ possiede, sicuramente si devono citare efficienza, versatilità e portabilità. L'efficienza di basso livello deriva proprio dal linguaggio C, da cui C++ eredita potenza e flessibilità, ed è stata sempre considerata un tratto essenziale, in quanto essa permette di usare il C++ per scrivere driver di dispositivi, o altro software che interagisce direttamente con l'hardware in tempo reale. In questo tipo di codice la prevedibilità delle prestazione è importante almeno quanto la velocità nuda e cruda, e spesso è importante anche la compattezza del risultato.

In generale, le ragioni per cui si usa C++, rispetto ad altri linguaggi di programmazione, sono le performance, la struttura a classi e oggetti, il riuso del codice, l'ereditarietà, i template. Oltre ad avere un'enorme Community di sviluppatori, dunque è possibile risolvere un problema a volte semplicemente documentandosi su internet, C++ a differenza di molti altri linguaggi di programmazione non appesantisce le prestazioni in fase di esecuzione. Questo permette di scrivere codice molto efficiente che ha comunque un alto livello di astrazione. Astrazioni come overloading, funzioni virtuali e puntatori non vengono "capite" dal compilatore, ma vengono tradotte a compile time, in modo da evitare overhead a runtime.

3.3 Perché usare Python

Python è un potente e flessibile linguaggio di programmazione ad alto livello con eccellenti capacità interattive e con una sintassi molto semplice. In questo caso però semplice non è sinonimo di impossibilità di realizzare software complessi. Anche se a primo impatto sembra essere realizzato per dummies e la sua semplicità lascia presagire che sia un linguaggio per imparare a programmare, Python grazie

al suo essere semplice offre la possibilità di tempi di sviluppo inferiori e concetti di strutture dati non fortemente tipizzati: questo permette la definizione di variabili come contenitori di qualsiasi tipo di dati, sarà l'interprete di Python che si occuperà della locazione di memoria e di tutte le caratteristiche da associare al dato.

Come un pitone, questo linguaggio è fluido, privo di sovrastrutture e di costrutti enormi da avere sempre in memoria. Si presta facilmente all'uso aziendale e di un possibile pacchetto Detector per ParkSmart. I Senior-Developer dell'azienda si potrebbero occupare di gestire il codice a basso livello, magari scritto in C++, mentre i nuovi arrivati, o Junior-Developer non avrebbero problemi a programmare in Python grazie ad un'interfaccia che comunica con il C++ stesso.

Python è stato progettato per essere estensibile a linguaggi a basso livello come C ed è estremamente utilizzato in ambito scientifico. A tal fine, librerie già esistenti o personalizzate scritte in C/C++ possono essere interfacciate a Python tramite estensioni.

3.4 Combinare C++ e Python

Python e C++ sono due linguaggi molto diversi sotto molti punti di vista: mentre C++ è normalmente compilato in codice macchina, Python è interpretato. La gestione dinamica dei tipi su Python è l'emblema della sua flessibilità, al contrario di C++ che gestisce i tipi in maniera statica, caposaldo della sua efficienza. La compilazione in C++ è un processo complicato e difficile mentre su Python avviene praticamente tutto a runtime. Eppure queste differenze per molti programmatori sono un vantaggio e fanno si che Python e C++ si completino perfettamente. Procedure lente che rappresentano dei colli di bottiglia in Python, possono essere riscritte in C++ per sfruttare la massima velocità e autori di librerie scritte in C++ scelgono Python come linguaggio Middleware per le sue capacità di integrazione di sistemi flessibili.

4

Studio di Tecnologie esistenti

Diverse sono le possibili soluzioni al problema dell'interoperabilità tra diverse interfacce. Molte persone hanno lavorato sull'integrazione di codice C++ con Python essendo un task complesso come suggerito dalla continua presenza del Python Special Interest Group for C++.

4.1 Cython

Le prime prove di wrapping sono state effettuate con Cython, sotto consiglio del team di ParkSmart. Anche se la documentazione è scarna sul web, molti nella comunità scientifica e sul web vantano questo meta-linguaggio di programmazione.

Il linguaggio di programmazione Cython, da non confondere con CPython, è un superset di Python con una funzione interfaccia per invocare routines di C/C++ con la possibilità di dichiarare subroutine statiche, variabili locali e attributi di classi. Cython è un linguaggio compilato che genera moduli estensivi di CPython. Questi moduli possono essere caricati usando normalmente Python grazie alla chiamata `import`. Un semplice programma in Cython è più complesso rispetto alla maggior

parte dei linguaggi perché si interfaccia con le Python C API e con le distutilis. Per un progetto base servono almeno tre file:

- un `setup.py` che invoca le distutils che generano il modulo di estensione.
- Il programma (main) per caricare il modulo estensione, che è generalmente scritto in `.pyx`.
- i codici sorgenti di Cython (scritti anche in C/C++).

La sintassi di Cython si avvicina a quella di Python ed è possibile integrare in un listato Cython codice C e Python tramite giuste definizioni. Sul web, Cython è più di Python:

- Cython converte codice Python in C, questo codice può essere compilato con un qualsiasi compilatore C e può produrre un eseguibile oppure una libreria dinamica.
- è possibile scrivere direttamente in Cython codice C e Python creando un ponte tra i due
- può espandere la sintassi sia di Python che di C mantenendo però la tipizzazione delle variabili.

La tipizzazione su Cython è una caratteristica fondamentale: poter dichiarare la variabile con un tipo incrementa di molto la velocità di esecuzione. Sono stati i benchmark visti in rete e testati a convincerci che Cython potesse fare al caso nostro, dato che il target finale era quello di mappare in memoria dati di grandi dimensioni, per poi poterli elaborare con le librerie di cui dispone ParkSmart, operazione sicuramente dispendiosa in termini computazionali.

Prima di cominciare lo studio, e le prove su codici personali, ho voluto testare quanto effettivamente fosse efficiente Cython ed i risultati sono stati straordinari.

E' possibile offrire una dimostrazione grazie ad un tool molto potente che ci permette di studiare questo tipo di esperimenti.

IPython offre una ricca architettura per la programmazione interattiva con:

- una potente shell interattiva
- un kernel per Jupyter
- supporta un interattiva visualizzazione e uso di GUI toolkits.
- semplice da usare, offre la possibile di lavorare in programmazione parallela con dei tool dedicati.

Grazie a questo tool riusciamo ad effettuare dei test per provare l'efficienza di Cython.

```
>>> load_ext cythonmagic

>>>         %%cython
                def fib(int n)
                        cdef int a, b, i
                        a, b = 1,1
                        for i in range(n):
                                a,b = a+b, a
                        return a

>>> fib(3)
out[] = 5

>>> %timeif fib(3)
        10000000 loops, best of 3: 67.9 ns per loop

>>> def pyfib(n):
        a, b = 1,1
        for i in range(n)
                a, b = a+b, a
        return a

>>> %timeit pyfib(3)
        1000000 loops, best of 3: 810 ns per loop
```

Con lo stesso codice, ma scritto in Cython, abbiamo avuto prestazioni circa 12 volte migliori. Diverse prove sono state effettuate con codici di prova. L'approccio è stato quello di scrivere un codice semplice e non molto elaborato in C++ che tendesse ad essere molto simile a quello del Detector di ParkSmart, in modo da evitare errori vari durante le prove.

```
#include <iostream>
#include <string>

namespace { // Avoid cluttering the global namespace.

  // A friendly class.
  class hello
  {
    public:
      hello(const std::string& country) { this->country = country; }
      std::string greet() const { return "Hello from " + country; }
    private:
      std::string country;
  };
}
```

Diversi test, ed a volte anche adattamenti del codice C++ a Cython, hanno meglio sottolineato quale era il target del progetto ed evidenziato quali sono gli effettivi svantaggi di Cython:

- **Cython è un metalinguaggio**, per funzionare in maniera efficiente si dovrebbe combinare C e Python, riscrivendo tutte le procedure del Detector in Cython: operazione costosa all'azienda, anche per future modifiche al sorgente del Detector stesso. Un futuro sviluppatore C++ per poter contribuire al Detector dovrebbe necessariamente studiare le dinamiche di Cython e scrivere pezzi aggiuntivi in Cython per esporre le nuove funzionalità aggiunte lato C++ in Python.

- **Cython è un progetto in via di sviluppo**; non c'è il supporto a librerie standard di C++ come std::string, fondamentale per lo sviluppo del Detector. Modificare Detector con tipi supportati da Cython vorrebbe dire re-ingegnerizzare tutto il sorgente.

- **Non offre la possibilità di creare un wrapping "automatico" per librerie C**, dunque nel caso in cui Detector avesse un gran numero di metodi pubblici da esporre si aumenterebbe di gran lunga il lavoro degli sviluppatori e di conseguenza la possibilità di incorrere in errori di vario tipo.

Sicuramente Cython, almeno per il presente, non fa al caso del progetto Detector di ParkSmart dato che non è possibile creare un'Adapter da manuale che

è effettivamente ciò che serve all'azienda. Dopo aver esposto il problema al team, è stata accolta la mia idea di procedere alla ricerca di nuove soluzioni che possano effettivamente creare un vero e proprio bridge tra Python e C++ mantenendo efficienza e usabilità.

4.2 SWIG

Un wrapping-tool che fa molto parlare è SWIG che crea estensioni C/C++ in maniera del tutto automatica. Da un file header, generalmente con estensione .h, è possibile generare una libreria che si può importare su Python. Un grande limite, che hanno diversi di questi tool, è il completo supporto a C a discapito del C++; purtroppo SWIG non supporta diverse funzionalità di C++ che vengono spesso usate, come overloading di metodi e operatori, eccezioni, namespace, distruttori privati e std::string. In generale non è molto apprezzato dalla comunità come funziona il parser di SWIG che, essendo un progetto ancora in via di sviluppo, risulta limitato. Un requisito per usare SWIG è quello di conoscere le C-API di Python per la conversione dei tipi.

4.3 Altri tool

Ci sono diversi progetti che hanno come obbiettivo quello di interfacciare librerie C++ a Python, la stessa wiki di Python ce ne propone alcuni tra i più famosi:

- CXX è una libreria per scrivere estensioni Python in C++, bisogna creare file bridge in .cxx, dunque è presente un metalinguaggio, con conseguenti difficoltà implementative.

- SIP è un tool per la generazione veloce di moduli Python per interfacciare librerie C/C++, è molto simile a SWIG, ma è stato sviluppato per generare moduli python. SIP richiede un set di file di configurazione (.sip) che descrivono le API e generano il codice. SIP supporta più sintassi rispetto a SWIG ma è comunque un tool di terze parti che ha bisogno di file specifici in un possibile progetto.

- AUTOWRAP è stato un tool che mi ha permesso di capire le limitazioni di Cython. Questo tool grazie ad una serie di processi automatizzati, genera codice Pyrex dato in input un file .pxd, che è genericamente un header molto simile ai file .h ma scritto in Cython. E' un progetto ancora in beta, e come Cython non supporta std::string.

- Py++ usa un parser per leggere codice C++ e generare codice con Boost.Python; è un tool automatico che non assicura prestazioni.

- Pybindgen genera codice lavorando con le C-API di Python. Si possono descrivere facilmente funzioni e classi in un file Python, oppure far generare automaticamente a Pybindgen un binding tra le interfacce, leggendo i file header (usando pygccxml, una libreria Python scritta dall'autore di Py++). E' comunque un progetto giovane, con un team piccolo. Ci sono delle limitazioni: non si possono esporre limitazioni C++, e non si può usare ereditarietà per le classi.

Diversi di questi progetti, e di altri non elencati, presentano un lavoro discontinuo che non garantisce scalabilità; addirittura molti di questi progetto sono stati scritti ad hoc per specifici utilizzi e di conseguenza sono poco flessibili ed affidabili.

Oltre a queste pesanti mancanze, l'uso di tool automatici crea ripetizione di codice e di conseguenza, perdita di efficienza.

5

La soluzione scelta: Boost.Python

Le limitazioni di tradizionali moduli di estensione tendono a contenere ripetizione di codice, che li rende poco efficienti. Da qui lo sviluppo di tool per wrapping. La maggior parte di essi introducono un loro linguaggio per il binding tra i linguaggi. Sicuramente questo porta vantaggi, ma avere a che fare con tre linguaggi di programmazione differenti (Python, C++ e il linguaggio di interfaccia o meta-linguaggio) produce difficoltà nell'implementazione. CXX ad esempio, evita l'introduzione di un terzo linguaggio, ma non include il supporto per classi C++.

Boost.Python è una libreria C++ open source che permette la creazione di interfacce per un binding di classi e metodi da C++ a Python. Sfruttando la potenza in compile-time di C++ e grazie allo sviluppo di recenti tecniche di meta-programmazione, Boost.Python è scritto in C++ puro, senza l'introduzione di nuova sintassi. Il ricco set di funzionalità e l'interfaccia ad alto livello di Boost.Python permettono di progettare con facilità pacchetti ibridi, dando semplicemente ai programmatori accesso all'efficienza del compilatore di C++ nella fase di compilazione e l'estrema convenienza dell'interprete a runtime di Python.

Le caratteristiche e gli obiettivi di Boost.Python sovrappongono in maniera significativa quelle di questi sistemi. Boost.Python tenta di massimizzare l'efficienza e la flessibilità senza introdurre un nuovo linguaggio, presentando all'utente un'interfaccia ad alto livello per esporre classi e funzioni C++. Boost.Python va oltre l'ambito dei sistemi precedentemente descritti fornendo supporto a funzioni virtuali, puntatori, references e overloading.

L'obiettivo primario di Boost.Python è quello di permettere agli utenti di esporre classi e funzioni C++ a Python usando solamente il compilatore, in modo da poter manipolare oggetti C++ da Python. Tuttavia, è anche importante non tradurre le interfacce troppo alla lettera. Nel caso del concetto di ereditarietà, presente sia in C++ che in Python in maniera diversa, Boost.Python è in grado di colmare il gap tra le interfacce.

Data la ricca interoperabilità delle C-API di Python, dovrebbe essere in linea di principio possibile esporre tipi e funzioni scritte in C++ per interfacce Python. Tuttavia i servizi forniti da Python per l'integrazione con C++ sono relativamente scarsi. Rispetto al C++ e Python, C dispone di strutture di astrazione molto rudimentali, e il supporto per la gestione delle eccezioni è completamente mancante.

Le limitazioni dei tool sopra descritti hanno portano all'individuazione dell'obiettivo da parte degli sviluppatori di Boost.Python, cercando di massimizzare la convenienza e la flessibilità senza di introdurre un terzo linguaggio, vuole dunque dare all'utente un interfaccia C++ ad alto livello per creare facilmente wrapping di classi e funzioni, cercando di gestire la complessità che c'è dietro le quinte con la meta-programmazione statica. Un altro obiettivo di Boost.Python è quello di andare oltre quello che gli altri sistemi fanno, come:

- supporto per le funzioni virtuali C++ che possono essere sovrascritte in Python.
- funzionalità di gestione per puntatori e riferimenti a basso livello.
- supporto per organizzare estensioni come pacchetti Python, con un registro centrale per le conversioni di tipi.

- Un meccanismo sicuro e conveniente per legare Python con un potente motore di serializzazione (pickle).

Grazie a questa potente libreria è possibile creare wrapping non intrusivi, senza modificare o addirittura vedere il codice C++.

5.1 Esporre Classi con Boost.Python

Il primo passo dopo l'individuazione di Boost.Python è stato quello di provare a testare con la classe che ci eravamo preposti di far funzionare già dall'esempio in Cython. Per poter generare un interfaccia di classi da C++ a Python con Boost.Python serve:

- installare Boost.Python sulla macchina.
- un file C++, dove generalmente sono definiti i metodi che si vogliono esporre.
- un file `setup.py` che richiamato in fase di compilazione utilizza le distutils di Python per generare il modulo di estensione.

Di seguito un esempio della classe che vogliamo esporre a Python.

```
1  #include <iostream>
2  #include <string>
3
4  namespace { // Avoid cluttering the global namespace.
5
6    // A friendly class.
7    class hello
8    {
9      public:
10       hello(const std::string& country) { this->country = country; }
11       std::string greet() const { return "Hello from " + country; }
12     private:
13       std::string country;
14   };
15 }
16
17 //Building the extension module
18 #include <boost/python.hpp>
19 using namespace boost::python;
20
```

```
21 BOOST_PYTHON_MODULE(getting_started)
22 {
23     // Create the Python type object for our
24     // extension class and define __init__ function.
25     class_<hello>("hello", init<std::string>())
26         .def("greet", &hello::greet)  // Add a regular member function.
27     ;
28 }
```

E' preferibile creare un nuovo namespace per evitare conflitti, e in seguito grazie alla chiamata `BOOST PYTHON MODULE` possiamo dichiarare il modulo e quali saranno i metodi da esporre, in genere solo i metodi pubblici delle classi. Questa sintassi è valida sia per le classi che per le struct.

5.1.1 Python Distutils

Le Distutils per Python (Python Module Distribution Utilities) sono sviluppate per soddisfare un bisogno a lungo termine della comunità di Python. Si vuole fornire un meccanismo standard per il building, la distribuzione e l'installazione di moduli Python, o meglio ancora, distribuzioni multi-modulo. Le Distutils per Python riescono a soddisfare questo bisogno provvedendo a fornire un set di classi per diverse esigenze (creare estensioni C, processare documentazione, installare in directory specifiche le librerie, compilare Python in bytecode, etc.). La necessità di tale progetto è chiara: si vuole fornire a chiunque abbia installato più di una distribuzione di moduli di grandi dimensioni per Python la possibilità di effettuare il build dei progetti Python sempre allo stesso modo, evitando difficoltà agli sviluppatori e agli utenti che eseguiranno i diversi moduli.

L'interfaccia delle Distutils per Python è fortemente influenzata da MakeMaker. Superficialmente Distutils si basa su uno script `setup.py` invece del `Makefile.PL`. A differenza del makefile, lo script consiste in una chiamata setup, tuttavia il meccanismo dietro a questa chiamata è tutto implementato in Python senza lo step di `make`. Questo vuol dire che l'azione di building o installazione viene soddisfatta immediatamente, inoltre l'utente deve specificare di cosa ha

bisogno nel comando per `setup.py`. La chiamata tipo per eseguire uno script setup basato su Distutils è:

```
python setup.py -v install
```

che tiene conto di tutto ciò che è richiesto per installare un modulo di distribuzione funzionante. Questo comando si occupa di:

- trovare moduli scritti in Python puro
- compilare e fare il link dei moduli di estensione C/C++
- compilare moduli di estensione Java
- processare documentazione per uno o più formati standard
- avviare suite di test
- installare tutti i file in posti sensibili (seguendo l'installazione di Python e le preferenze dell'utente).

Di seguito un esempio testato di come è possibile compilare la classe `hello` con Boost.Python invocando le Distutils:

```
1 from distutils.core import setup
2 from distutils.extension import Extension
3
4 setup(name="PackageName",
5     ext_modules=[
6         Extension("getting_started", ["hello.cpp"],
7         libraries = ["boost_python"])
8     ])
```

6
Conclusioni

Con la soddisfacente finalizzazione di questo progetto aziendale si riesce a dare la possibilità di sviluppare features in maniera semplice senza il requisito di conoscere C++ alla perfezione. Con la soluzione sviluppata si è altresì non obbligati a studiare pagine e pagine di documentazione.

L'achievement non è stato solo quello di fornire il mezzo agli Junior Developer, ma anche quello di creare una via semplificata ai Senior: ParkSmart potrà così arricchire le proprie librerie scritte in C++ ed esporle semplicemente modificando lo stesso file su cui è presente il modulo Python, inserendo una linea di codice come da template.

Durante lo sviluppo ho vissuto l'ambiente aziendale: sono stato continuamente spronato a fare meglio dal team e dal CTO di ParkSmart Giuseppe Patanè. Questo ha portato ad un'enorme nuova conoscenza che mi servirà per tutta la vita. Ho imparato nuove tecniche che continuo ad usare ogni giorno nei miei progetti accademici e personali. Riesco a trovare soluzioni a problemi in maniera più veloce ed efficiente perché ho imparato un giusto modo di affrontare problemi informatici.

L'essere catapultati in un ambiente di lavoro porta a comprendere che lo studio e

il metodo è fondamentale per risolvere particolari problemi. Il non aver mai lavorato con Python o con altre tecnologie presentate in questa tesi, sicuramente mi ha messo in difficoltà, ma con la giusta determinazione sono riuscito ad ottenere risultati inaspettati.

Bibliografia

[1] **Design Pattern: Adapter**, www.en.wikipedia.org/wiki/Adapter_pattern

[2] **Cross-compilazione**, www.wiki.osdev.org/GCC_Cross_Compiler

[3] **Documentazione C++**, www.cplusplus.com

[4] **Documentazione Python**, www.python.org

[5] **Cython - C-Extensions for Python**, www.cython.org

[6] **SWIG - Simplified Wrapper and Interface Generator**, www.swig.org

[7] **CXX**, www.cxx.sourceforge.net

[8] **SIP**, www.riverbankcomputing.com/software/sip/intro

[9] **AUTOWRAP**, www.pypi.python.org/pypi/autowrap

[10] **Py++**, www.pyplusplus.readthedocs.io/en/latest/

[11] **Pybindgen**, www.pypi.python.org/pypi/PyBindGen

[12] **Boost.Python**, www.boost.org

[13] **Python Distutils**, www.docs.python.org/2/distutils/