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

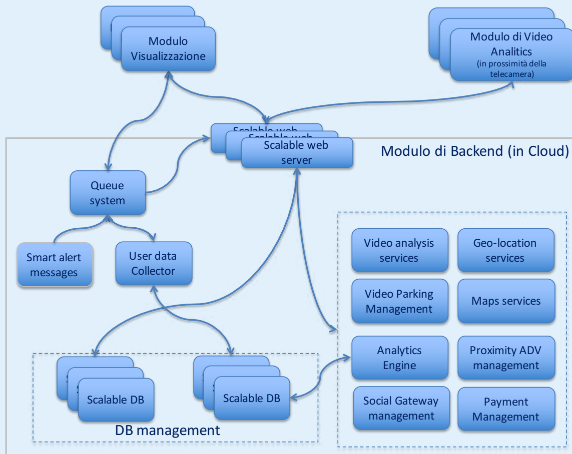
RELATORE
Chiar.mo Prof. Salvatore Riccobene

CORRELATORE
Dott. Giuseppe Patanè

Giuseppe Astuti

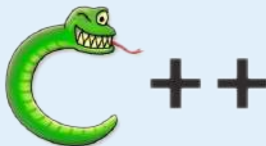
Relazione Progetto finale, 28 Luglio 2016

ParkSmart



Cross-Compilazione

Obiettivo



Fornire un mezzo di comunicazione tra C++ e Python in modo da sfruttare:

- La potenza ed efficienza del C++
- La semplicità di Python

Vantaggi di Python e C++

- I sorgenti di ParkSmart non verrebbero modificati
  - Librerie longeve - LTS
  - Minori risorse da utilizzare
  - Più efficienza computazionale
- Marcia in più in azienda
  - Junior Developer impegnati con Python
  - Senior Developer rilasciano aggiornamenti della libreria C++

Possibili soluzioni

Possono essere così classificate:

- Metalinguaggi
  - *Cython*, *CXX*
- Wrapping Tools
  - *SWIG*, *SIP*, *AUTOWRAP*

Metalinguaggi

Pro

- Si riuscirebbe a mantenere efficienza computazionale.

Contro

- Cython è un metalinguaggio scritto per C.
  - Non prevede alti livelli di astrazione
  - Librerie standard non supportate
- Per ogni nuova funzione in Libreria bisogna aggiornare anche il sorgente Cython
- Sintassi diversa da Python e C++.

Wrapping Tools

Problema comune

- Tools scritti per uno specifico utilizzo.
  - Lavoro di sviluppo discontinuo
  - Poco flessibili ed affidabili
- Ripetizione di codice

Migliore Soluzione testata

- Libreria scritta in C++ puro
- Crea interfacce di binding di classi e metodi
- Si sfrutta la potenza in compile-time di C++

Boost.Python - Esporre classi e metodi

```
1   #include <iostream>
2   #include <string>
3
4   namespace {         // Avoid cluttering the global namespace.
5     class hello {    // A friendly class.
6       public:
7         hello(const std::string& country) { this->country = country; }
8         std::string greet() const { return "Hello from " + country; }
9       private:
10        std::string country;
11    };
12  }
13
14  #include <boost/python.hpp>     //Building the extension module
15  using namespace boost::python;
16
17  BOOST_PYTHON_MODULE(getting_started){
18      // Create the Python type object for our extension class and define __init__ function.
19      class_<hello>("hello", init<std::string>())
20        .def("greet", &hello::greet);    // Add a regular member function.
21  }
```

Boost.Python - Requisiti

- Boost.Python installato sulla macchina
- Sorgente C++ con `BOOST_PYTHON_MODULE`
- file `setup.py` che utilizza le distutils per generare il modulo Python

```
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       ])
```

Conclusioni e obiettivi raggiunti

- Semplificazione dello sviluppo software
- Velocizzare la prototipazione sperimentale
- Facilitare l'inserimento di nuovi sviluppatori in azienda

Sviluppi futuri

- Integrazione nel processo di compilazione e test aziendale