Plug-in Based Software Architecture for Robotics 🤖

ABISHALINI SIVARAMAN

Cppcon
The C++ Conference

PICKNIK

2023
October 01 - 06

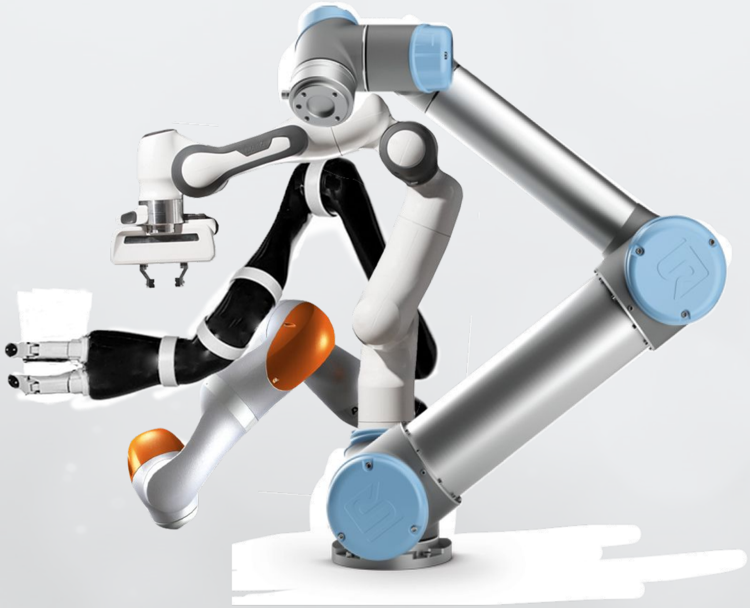
Outline

- What is plugin architecture?
- Why use plugin architecture?
- Designing a simplified plugin architecture
- Library used in robotics to implement plugin based system
  - Pluginlib
- Case study for plugin architecture - MoveIt
- Limitations
- Summary

PICKNIK

Introduction

- Abi Sivaraman
- Robotics Engineer at PickNik Robotics
- I work with robotic arms
- MoveIt Maintainer



What is plugin architecture?

Software Design Pattern that allows for developers to add functionality to a larger system without having to alter the source code of the system itself.

Plug-ins are self-contained modules that are loaded into the system at runtime.

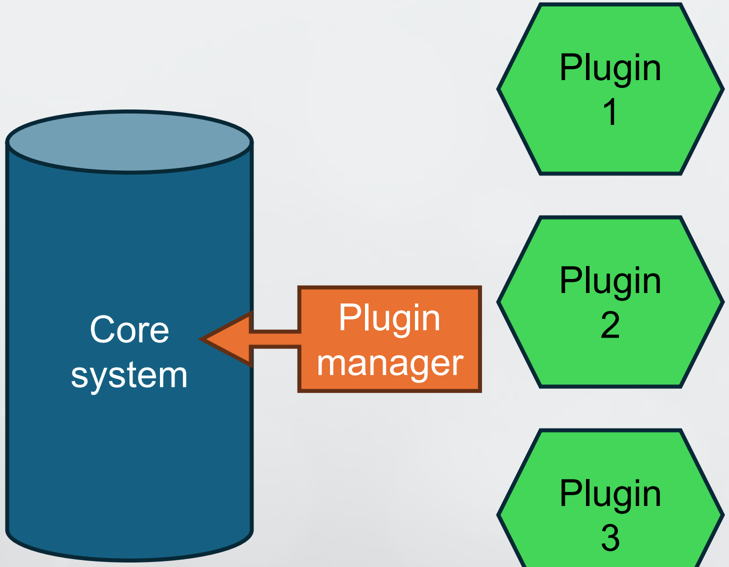
PICKNIK

Components of plugin architecture

1. **Core** - Defines how the system operates and provides interface

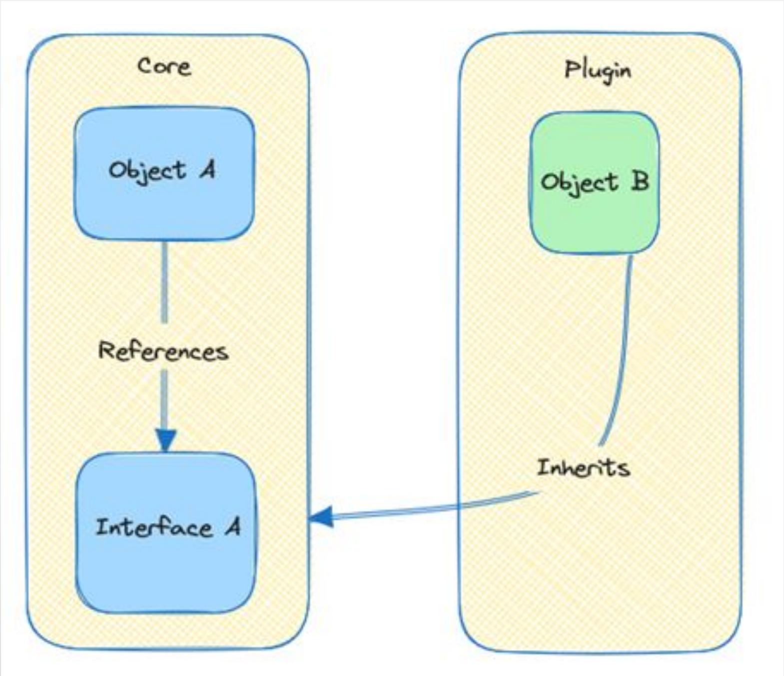
2. **Plugins** - Stand-alone independent components that contain implementation of core application's features

3. **Plugin Manager** - Component responsible for loading and unloading plugins



Why use plugin architecture?

- You do not want to nail down a specific implementation
  - Dependency Inversion



- Modular code
  - Can help organize large projects
  - Reduce dependencies in the core system

Why use plugin architecture?

- More testable code
  - Interface can be mocked to test core system
  - Plugins can be individually tested

- Simplifies integration
  - Core system has a well defined API
  - Increase collaboration as a side effect

Why use plugin architecture?

- Do not have to compile the entire project

- Plugins are be loaded, updated and deleted on the fly without application restart

Some popular C++ projects that use plugins

- Audio editing software
- Image editing software
- Text Editors and IDE
- Game Engines

Designing the components of the plugin architecture

Core System

Let us design a system that will move robot arms using the plugin architecture.

Required functionality

- Given start and end robot state, create a motion plan

PICKNIK

Core System contains the Interface class

```cpp
#pragma once
#include <vector>
#include <string>

namespace motion planner
{

struct RobotState {
    std::vector<std::string> joint_names;
    std::vector<double> joint_values;
};

class IMotionPlanner
{
public:
    virtual ~IMotionPlanner() = default;
    virtual bool initialize() = 0;
    virtual std::vector<RobotState> plan (RobotState start, RobotState goal) = 0;
};
}
```

This is an

abstract class

PICKNIK

Core System contains the Interface class

```
#pragma once
#include <vector>
#include <string>

namespace motion planner
{

struct RobotState {
    std::vector<std::string> joint_names;
    std::vector<double> joint_values;
};

class IMotionPlanner
{
public:
    virtual ~IMotionPlanner() = default;
    virtual bool initialize() = 0;
    virtual std::vector<RobotState> plan (RobotState start, RobotState goal) = 0;
};
}
```

The Robot State comprises of the joint names and their position values

Core System contains the Interface class

```cpp
#pragma once
#include <vector>
#include <string>

namespace motion planner
{

struct RobotState {
    std::vector<std::string> joint_names;
    std::vector<double> joint_values;
};

class IMotionPlanner
{
public:
    virtual ~IMotionPlanner() = default;
    virtual bool initialize() = 0;
    virtual std::vector<RobotState> plan (RobotState start, RobotState goal) = 0;
};
}
```

The plan function

Plugin contains the Implementation Class

```
#include "motion planner interface.hpp"

class SimpleMotionPlanner : public IMotionPlanner
{
public:
    SimpleMotionPlanner() = default;
    ~SimpleMotionPlanner() = default;


    std::vector<RobotState> plan(RobotState start, RobotState goal) override
    {
        std::vector<RobotState> motion_plan = {};
        // Insert a simple motion planner
        return motion_plan;
    }
};
```

Plugin contains the Implementation Class

```
#include "motion planner interface.hpp"

class SimpleMotionPlanner : public IMotionPlanner
{
public:
    SimpleMotionPlanner() = default;
    ~SimpleMotionPlanner() = default;


    std::vector<RobotState> plan(RobotState start, RobotState goal) override
    {
        std::vector<RobotState> motion_plan = {};
        // Insert a simple motion planner
        return motion_plan;
    }
};
```

The SimpleMotionPlanner inherits from IMotionPlanner

Plugin contains the Implementation Class

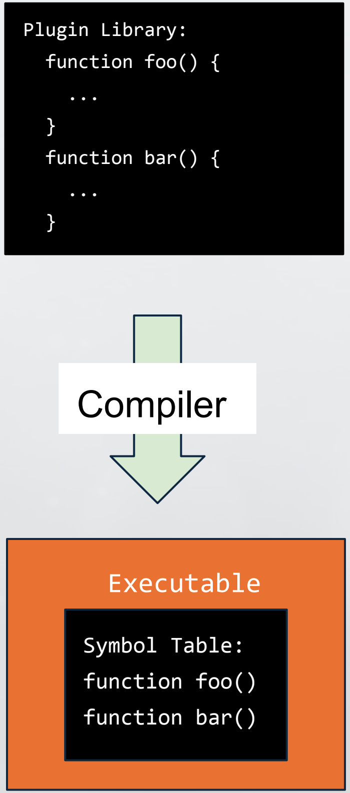
```cpp
#include "motion planner interface.hpp"

class SimpleMotionPlanner : public IMotionPlanner
{
public:
    SimpleMotionPlanner() = default;
    ~SimpleMotionPlanner() = default;

    std::vector<RobotState> plan(RobotState start, RobotState goal) override
    {
        std::vector<RobotState> motion_plan = {};
        // Insert a simple motion planner
        return motion_plan;
    }
};
```

Add your simple motion planner

PICKNIK

Building the plugin

- Should be built as a shared library
- As a result, the file should have the following extension
  - .so in UNIX based system
  - .dll in Windows
  - .dylib in MacOS



How to use the plugin in the core system

- This is the role of the Plugin Manager

- Concept: Dynamic Loading
  - Load library into memory at runtime
  - Retrieve addresses of functions
  - Execute those functions
  - Unload the library from memory when not in use

PICKNIK

Task of Plugin Manager - Part 1

- Determine the path to the shared libraries
  - Prefix - folder where the library was installed to
  - Shared library name
  - Suffix - extension dependent on OS

Task of Plugin Manager - Part 2

- Loading and unloading shared libraries given library path

- Operating system dependent
    - dlopen/dlclose system calls in UNIX systems
    - LoadLibrary/FreeLibrary API calls in Windows

Loading and Unloading libraries in UNIX systems

```
#include <dlfcn.h>
class SharedLibrary
{
public:
 Shared Library(const std: :string& library_path)
{
    library_path_ = library_path;
}

void load_library()
{
    handle_ = dlopen(library_path_, RTLD_LAZY);
}

void unload_library()
{
    dlclose(handle_) ;
}

private:
    std::string library_path_;
    void* handle_;
};
```

- GNU C Library
- Refer Linux Manual Page
- dlopen() loads the dynamic shared object given a string filepath

Loading and Unloading libraries in UNIX systems

```
#include <dlfcn.h>
class SharedLibrary
{
public:
 Shared Library(const std: :string& library_path)
{
    library_path_ = library_path;
}

void load_library()
{
    handle_ = dlopen(library_path_, RTLD_LAZY);
}

void unload_library()
{
    dlclose(handle_) ;
}

private:
    std::string library_path_;
    void* handle_;
};
```

- dlopen() returns an opaque “handle” for the loaded object which can be used by other function like dlclose()

Task of Plugin Manager - Part 3

- Creating an instance of the implementation class
- We now have loaded the shared object into memory using dlopen().
- We can use dlsym() to obtain address of a function in the shared object
  - GetProcAddress in Windows

PICKNIK

Add function in implementation class to create instance of implementation class

```cpp
class SimpleMotionPlanner : public IMotionPlanner
{
public:
    SimpleMotionPlanner() = default;
    ~SimpleMotionPlanner() = default;

    std::vector<RobotState> plan(RobotState start, RobotState goal) override
    {
       ….
    }
};

extern "C" IMotionPlanner* createInstance() { return new SimpleMotionPlanner; }
extern "C" void deleteInstance(IMotionPlanner* motion_planner) {delete motion_planner; }
```

Factory methods

PICKNIK

Why are they C functions?

```
extern "C" IMotionPlanner* createInstance() { return new SimpleMotionPlanner; }
extern "C" void deleteInstance(IMotionPlanner* motion_planner) {delete motion_planner; }
```

- extern “C” keyword makes them C functions
- Prevents the C++ compiler from mangling the name of the function so we can find the function symbols

Original Name	Mangled Name	Unmangled name
void generic_function(int a, int b)	_Z16generic_functionii	generic_function
void generic_function(float a)	_Z16generic_functionf	generic_function

Why are they C functions?

```
extern "C" IMotionPlanner* createInstance() { return new SimpleMotionPlanner; }
extern "C" void deleteInstance(IMotionPlanner* motion_planner) {delete motion_planner; }
```

- extern “C” keyword makes them C functions
- Prevents the C++ compiler from mangling the name of the function so we can find the function symbols

Original Name	Mangled Name	Unmangled name
void generic_function(int a, int b)	_Z16generic_functionii	generic_function
void generic_function(float a)	_Z16generic_functionf	generic_function

Use nm -gD <shared_object_name> to see the mangled names!

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
                        reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

create_motion_planner_instance is a function pointer. This can point to any function that takes in no arguments and return a pointer to an IMotionPlanner object.

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD_LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
                         reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

Make sure we have loaded the shared library into memory

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

Using dlsym, we search for the createInstance function in the shared library

The result of dlsym() or GetProcAddress() must be converted to a pointer of the appropriate type before it can be used

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

The result of dlsym() or GetProcAddress() must be converted to a pointer of the appropriate type before it can be used

PICKNIK

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
                        reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

Call the create_instance function which will use the C function we defined in the implementation class.

PICKNIK

Creating an instance of the implementation class using dlsym

```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
                        reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

Now you can call the plan function

PICKNIK

Creating an instance of the implementation class using dlsym

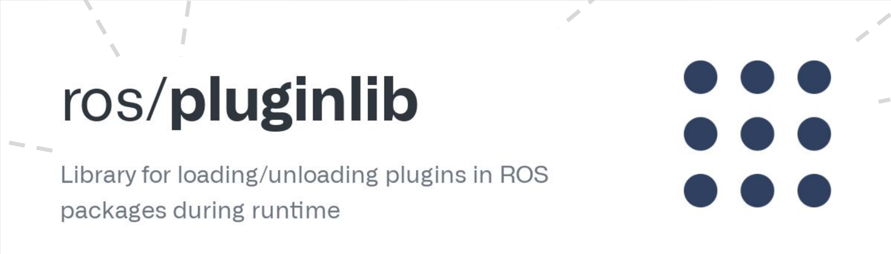
```
// create simple motion planner instance is a function pointer which points to any function that takes
// in no arguments and returns a IMotionPlanner type.
using create_motion_planner_instance = IMotionPlanner *(*)() ;

// Open shared library
void *simple_motion_planner_so = dlopen("/path/to/simple_motion_planner.so", RTLD LAZY);

// Use dlsym to obtain address of function createInstance. This result is then casted to type create_simple_motion_planner_instance
std::function<IMotionPlanner*()> create_instance =
                        reinterpret_cast<create_motion_planner_instance>(dlsym(simple_motion_planner_so, "createInstance"));

// Call the create instance function which returns a SimpleMotionPlanner raw pointer
IMotionPlanner* motion_planner = create_instance();

// Unload the shared object from memory
dlclose(simple_motion_planner_so);
```

Remember to call the deleteInstance function before unloading the shared library from memory

How complex can your Plugin Manager get

- Keep track of all the plugins loaded into memory
  - Plugin registry
- Work with static libraries too
  - Providing default implementation with core system
- Cross-platform
- Memory Management

Library used to implement plugin architecture in ROS projects



What is ROS?

- Robot Operating System
- Set of libraries and tools to help build robotic applications
- Open source
- Pub-sub architecture
- Launch system which starts nodes that can communicate between nodes using topics

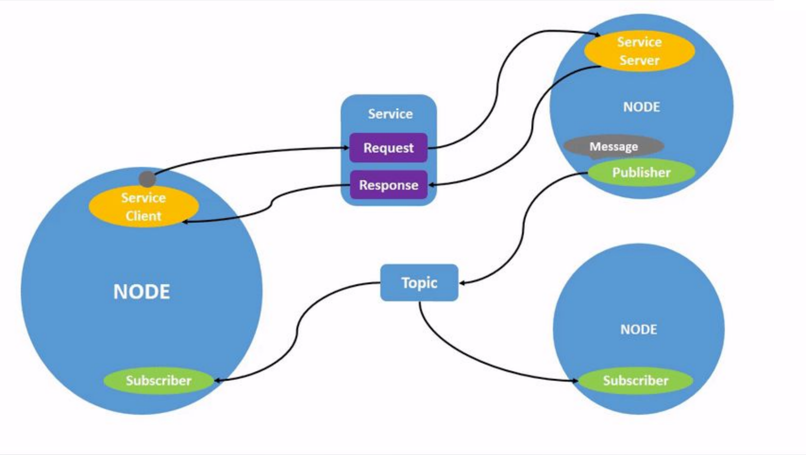


Image from https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html

Why use plugins over nodes?

- Network layer communication delays between nodes
- Need plugins for performance
- In a perfect world, ROS would not require plugins

Pluginlib

- Defacto Plugin Manager for ROS projects
- Built on top of class_loader, rcpputils and rcutils which takes care of system level calls
- Provides high level functions to be called in the core system.
- Cross platform

PICKNIK

Coming back to our system for motion planning

- Interface – `IMotionPlanner` ✅
- Implementation Class – `SimpleMotionPlanner` ✅

How do we load the Simple Motion Planner plugin using pluginlib?

PICKNIK

Register the plugin

```
class SimpleMotionPlanner : public IMotionPlanner
{
public:
    SimpleMotionPlanner() = default;
    ~SimpleMotionPlanner() = default;

    std::vector<RobotState> plan(RobotState start, RobotState goal) override
    {
      ....
    }
};

PLUGINLIB_EXPORT_CLASS(motion_planner::SimpleMotionPlanner ,
motion_planner::IMotionPlanner)
```

- Add macro at the end of the implementation class
- The macro takes base and derived class types as arguments
- Macro to create the factory method

PICKNIK

What does the macro call

```
#define CLASS_LOADER_REGISTER_CLASS_INTERNAL_WITH_MESSAGE(Derived, Base, UniqueID, Message) \
    namespace\
    { \
    struct ProxyExec ## UniqueID \
    { \
    type Derived _derived; \
    typedef Base _base; \
    ProxyExec ## UniqueID() \
    { \
        if (!std::string(Message).empty()) { \
            CONSOLE_BRIDGE_logInform("%s", Message); \
        } \
        holder = class_loader::impl::registerPlugin<_derived, _base>(#Derived, #Base); \
    } \
    private: \
        class_loader::impl::UniquePtr<class_loader::impl::AbstractMetaObjectBase> holder; \
    };\
    static ProxyExec ## UniqueID g_register_plugin_ ## UniqueID; \
    }
```

Creates a static object of type ProxyExecUniqueID
is a concatenation operator

PICKNIK

What does the macro call

```
#define CLASS_LOADER_REGISTER_CLASS_INTERNAL_WITH_MESSAGE(Derived, Base, UniqueID, Message) \
    namespace\
    { \
    struct ProxyExec ## UniqueID \
    { \
    type Derived _derived; \
    typedef Base _base; \
    ProxyExec ## UniqueID() \
    { \
        if (!std::string(Message).empty()) { \
            CONSOLE_BRIDGE_logInform("%s", Message); \
        } \
        holder = class_loader::impl::registerPlugin<_derived, _base>(#Derived, #Base); \
    } \
    private: \
        class_loader::impl::UniquePtr<class_loader::impl::MetaObject> holder; \
    };\
    static ProxyExec ## UniqueID g_register_plugin_ ## UniqueID; \
    }
```

Calls the function registerPlugin in the constructor
The # operator here stringifies the type

PICKNIK

Register the plugin

`registerPlugin` is a free function in ClassLoader

```
template<typename Derived, typename Base>
std::unique_ptr<MetaObject> registerPlugin(const std::string& class_name, const std::string&
base_class_name)
{
    // Make sure the library is loaded
    auto new_factory =
            std::make_unique<MetaObject<Derived, Base>>(class_name, base_class_name);
}
```

We have created a Metaobject object using the base and derived class details.
Let’s investigate what Metaobject holds.

MetaObject

MetaObject contains the factory functions (i.e.) to create and destroy instance of the Derived class.

```
template<class Derived, class Base>
class MetaObject
{
public:
    MetaObject(const std::string & class_name, const std::string & base_class_name)
    {}

    Base * create() const
    {
        return new Derived;
    }
};
```

This way, we don’t need to add the C function to our implementation class

PICKNIK

Keep track of all the plugins registered - Plugin registry

There is a static map variable in the ClassLoader to keep track of all the classes already registered

```
typedef std::string ClassName;

static std::map<ClassName, MetaObject *> factory_map;
```

All the factory methods are stored here

How does PluginLib figure out the path of the shared libraries

- Now that we have registered the plugin classes, we need to load the share object into memory
- Pluginlib parses XML files to get information on plugin name, shared library name, base and derived class type.

```xml
<library path="simple_motion_planner_plugin">
  <class name= "simple_motion_planner" type="motion_planner::SimpleMotionPlanner"
  base_class_type="motion_planner::IMotionPlanner">
    <description>
      Simple Motion Planner for Robot Arm
    </description>
  </class>
</library>
```

Core System loading Motion Planner

```cpp
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

Core system has a MotionPlan class which will load a Motion Planner and invoke its plan function

PICKNIK

Core System loading Motion Planner

```cpp
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

The input argument is the package name to find the location of the XML file

PICKNIK

Core System loading Motion Planner

```
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

On construction of ClassLoader object, a dlopen system call will be made to load the plugin into memory

PICKNIK

Core System loading Motion Planner

```cpp
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

The input argument here is the name of the plugin specified in the XML file.

Core System loading Motion Planner

```cpp
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

The createUniqueInstance will invoke the factory function (create) to return an instance of the Derived class

Core System loading Motion Planner

```cpp
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader = std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner = loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

Now we can call the plan function that we implemented in the SimpleMotionPlanner class

PICKNIK

Core System loading Motion Planner

```
#include <memory>
#include <motion_planner_interface.hpp>
#include <class_loader.hpp>

class MotionPlan
{
public:
    std::vector<RobotState> createMotionPlan(RobotState start, RobotState goal)
    {
        // Load the correct planner. What should the planner be? In ROS world, we send the package name and base class
name.
        // So, we need something to find the path of .so and the interface name
        using LoaderType = pluginlib::ClassLoader<IMotionPlanner>
        std::shared_ptr<LoaderType> loader =
                    std::make_shared<LoaderType>("package_name");

        // In ROS world, we send the plugin name which is declared in the XML file. Maybe we send the derived class name?
        std::shared_ptr<IMotionPlanner> motion_planner =
                            loader->createUniqueInstance("simple_motion_planner");

        // Call the plan function
        return motion_planner->plan(start, goal);
    }
}
```

Can be made ROS2 parameters

The input arguments here are read from a YAML file when the system is first started

And it can be modified during runtime

Can change out plugins

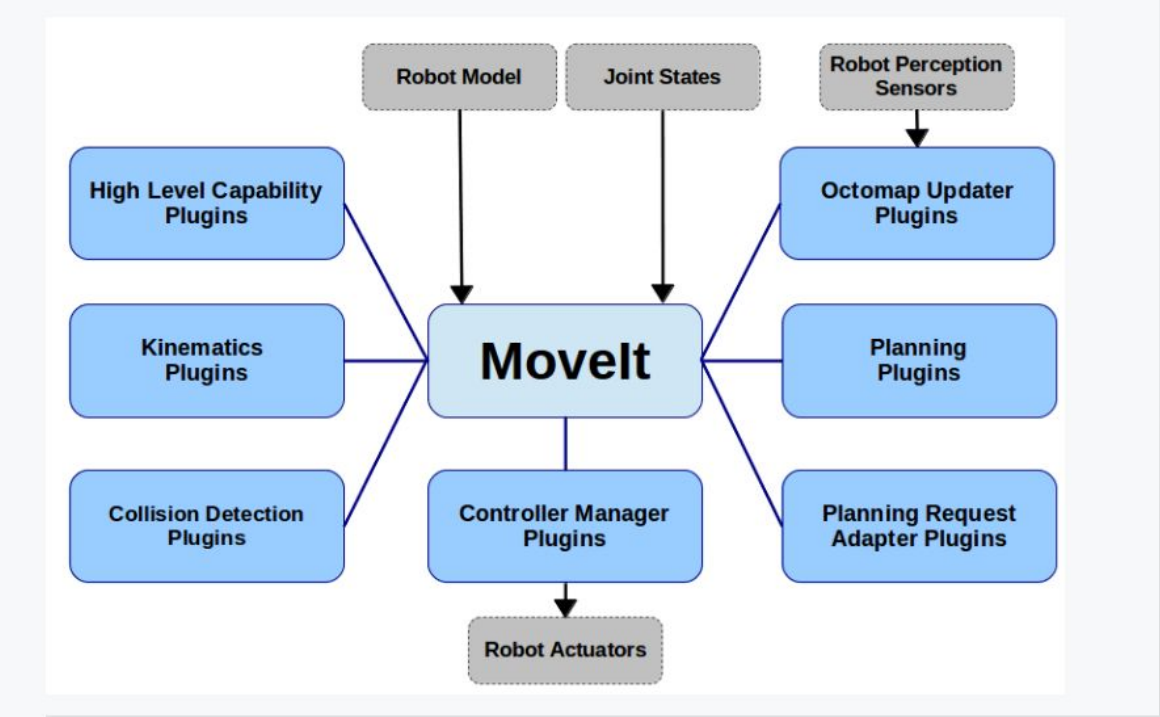
Summarizing how pluginlib made it easy to register and use plugins

- Add EXPORT macro in implementation file
- Add an XML file which contains library name, plugin name, base class type and derived class type
- Add `pluginlib_export_plugin_description_file(<>.xml)` so when during build, the xml file is in a known location.
- In core system, create ClassLoader instance and call the createUniqueInstance function.

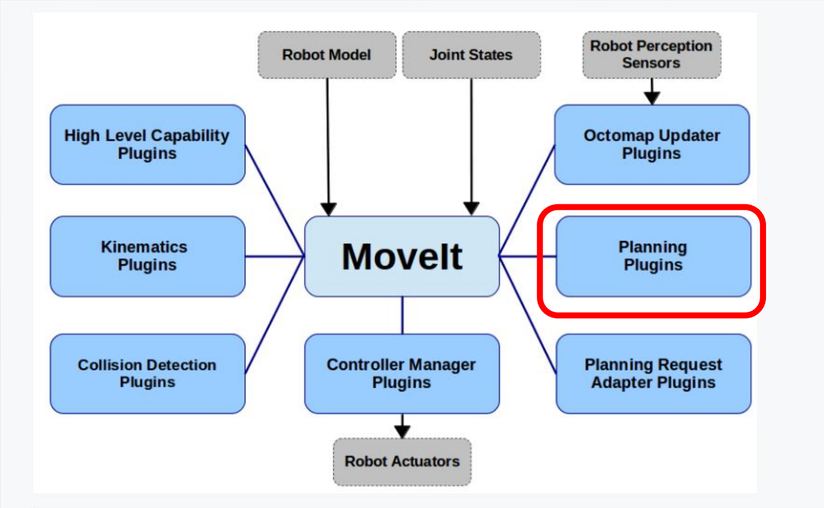
A much more complicated Motion Planning framework - MoveIt

- Open source robotics manipulation platform
  - Motion Planning
  - Grasping
  - 3D Perception
  - Robot control
  - Kinematics
- Built on ROS/ROS2
- Uses plugins extensively

Plugins in MoveIt



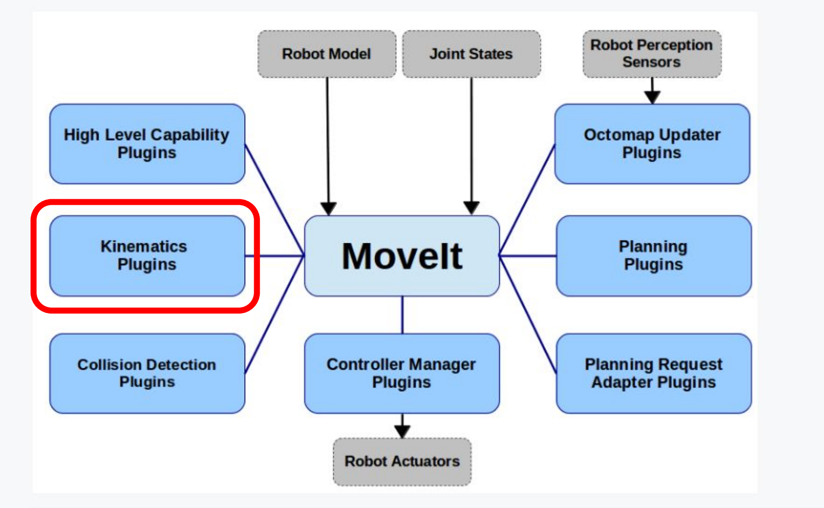
Motion Planning



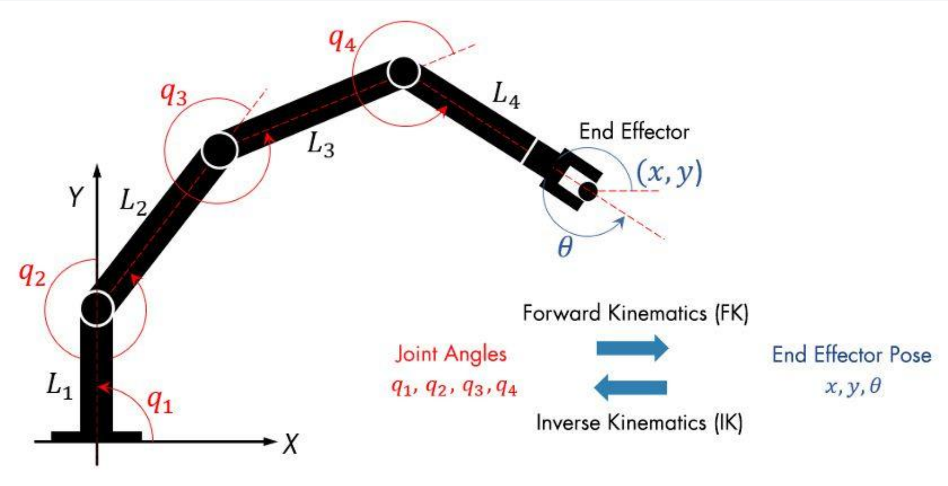
Plugins built on top of

- Open Motion Planning Library (OMPL)
  - Maintained by Kavraki Lab at Rice University
- Pilz
  - Pilz GmbH & Co.
- STOMP
  - Research paper from ICRA

Kinematics

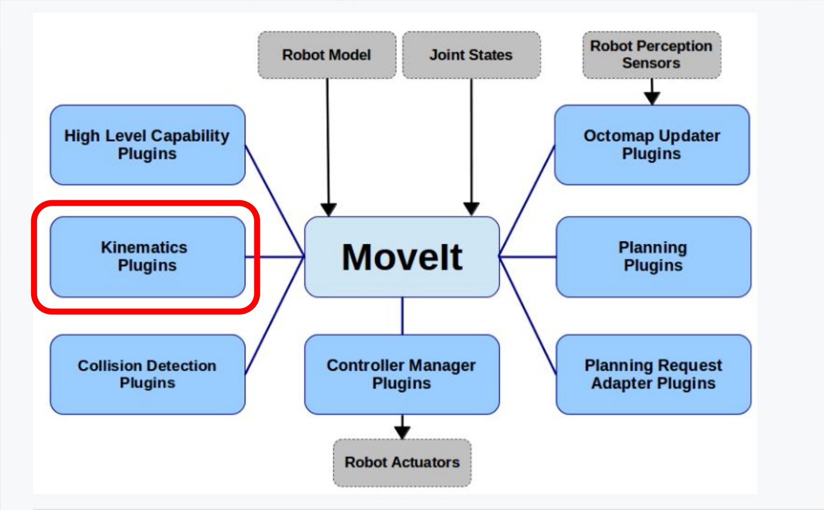


Robot Model
Joint States
Robot Perception Sensors
High Level Capability Plugins
Octomap Updater Plugins
Kinematics Plugins
MoveIt
Planning Plugins
Collision Detection Plugins
Controller Manager Plugins
Planning Request Adapter Plugins
Robot Actuators



q_4
q_3
L_4
L_3
End Effector
(x, y)
Y
L_2
θ
q_2
Forward Kinematics (FK)
Joint Angles
q_1, q_2, q_3, q_4
End Effector Pose
x, y, θ
L_1
q_1
Inverse Kinematics (IK)
X

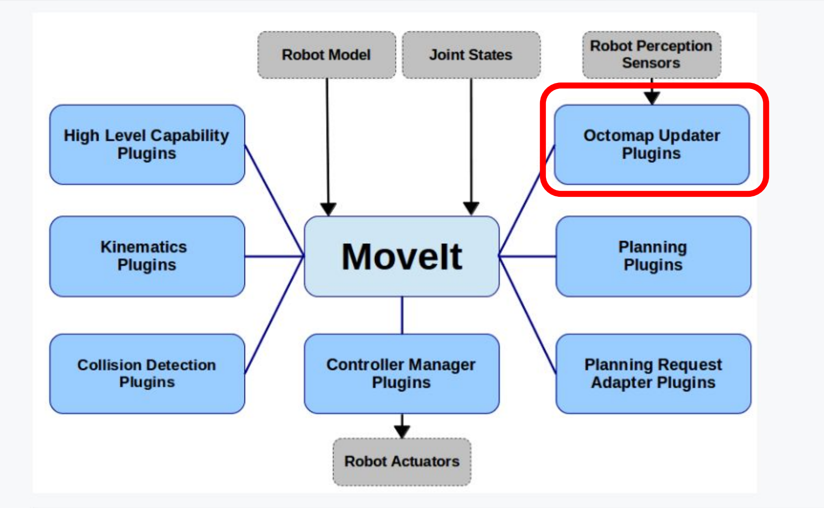
Kinematics

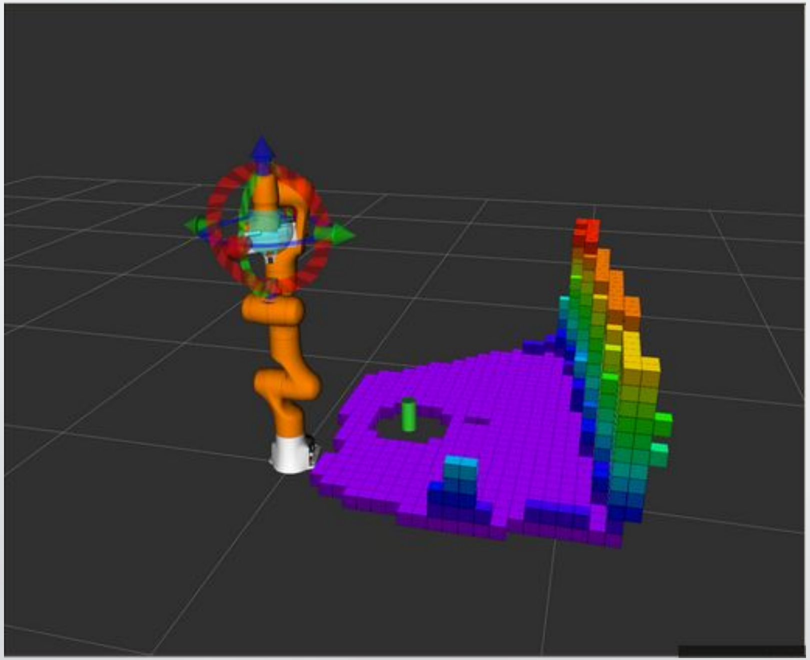


Plugins built on top of

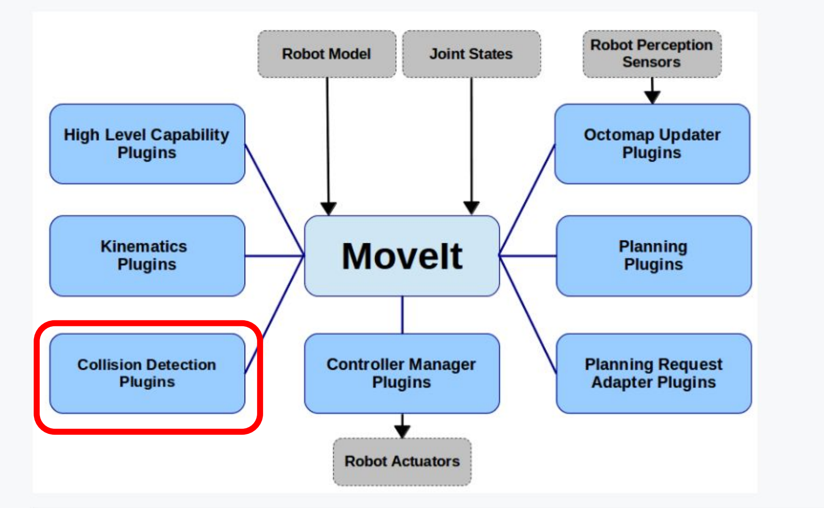
- TrackIK
  - TRACLabs, Inc. Robotics and Automation
- KDL
  - Kinematics and Dynamic Library
- BioIK
  - Philipp Ruppel as part of his Master Thesis
- PickIK
  - Developed and maintained by PickNik Robotics

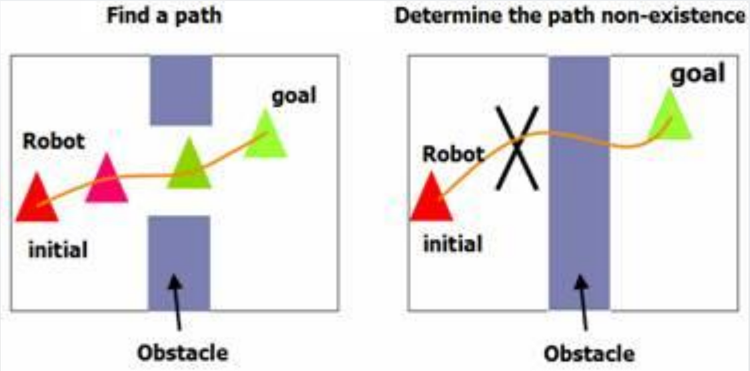
3D scene updater



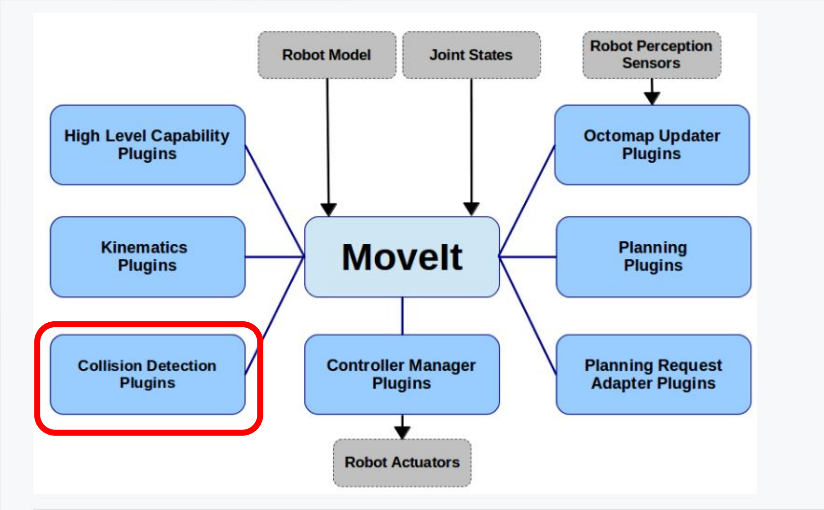


Collision Detection





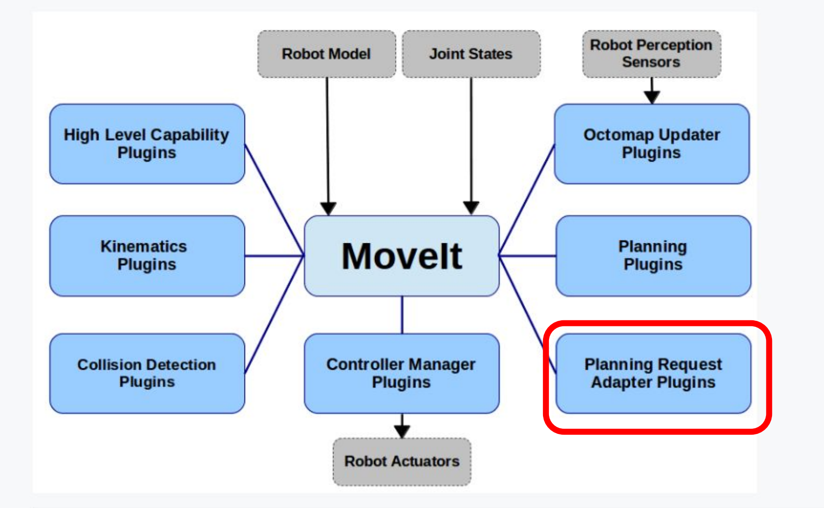
Collision Detection



Plugins built on top of

- FCL
  - Flexible Collision Library
- Bullet Physics SDK
  - Main author - Erwin Coumans

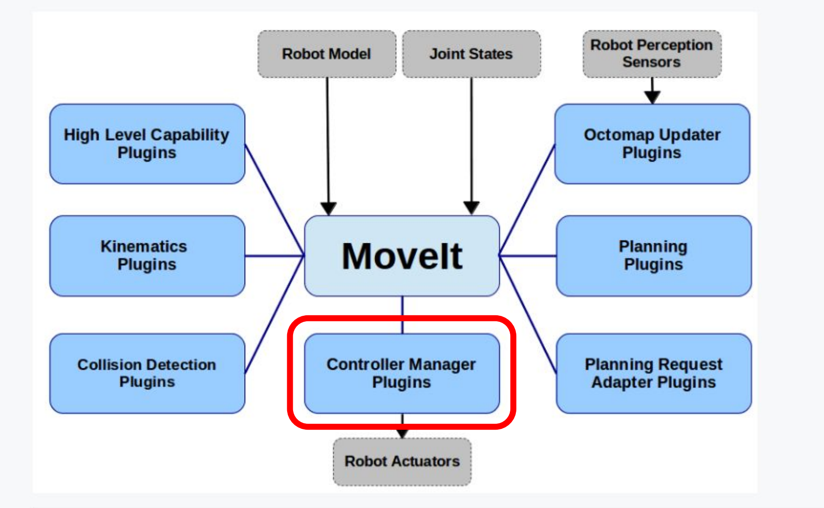
Planning Request Adapters



Trajectory Processing

- Fix Start State in Collision
- Fix workspace bounds
- Trajectory Smoothing

Controller Manager



- Robot Control using
  - Positions
  - Velocity
  - Effort

Libraries that can help build your plug in system

- Boost.DLL
  - Headers only library
  - Low level API through shared_library class
  - Tutorials
- POCO
  - SharedLibrary class
- Qt Plugins
  - Expand Qt UI application.

PICKNIK

Limitations of plugin architecture

- Plugins have to compiled with the same or compatible compiler as the core system
    - Different compilers and different versions of the same compiler can have different mangling techniques

- API needs to be fairly stable
    - Plugins should use standardized interfaces

Limitations of plugin architecture

- Testing individual plugins is easy but it is unknown how multiple plugins working together will affect the system

Example from MoveIt -

- Fix Start State in Collision
- Fix workspace bounds

PICKNIK

Limitations of plugin architecture

- Security issues
  - Even a well-meaning plugin may contain a bug which can crash the system or leak memory

Takeaway

- Plugin architecture - Loading modules at runtime
- When should you use this architecture?
- Important concepts
  - Runtime polymorphism
  - Dynamic Loading

Takeaway

- How to create your own plug-in based system
- Pluginlib
  - Library used by ROS projects to create a plug-in system
- Plugins used in MoveIt
- Drawbacks

Code available at

https://github.com/Abishalini/cppcon2023

Shout out to Anthony Baker and Griz Brooks!

The repo contains an implementation of plugin architecture using modern C++ and other best practices!

You can contribute!

- All libraries discussed are open source
- Those who want to get started with robotics
  - Bring your C++ skills!
  - Bring your own robot!

MoveIt

PICKNIK

PickNik Robotics -
The Unstructured Robotics Company

- Check out the company that maintains MoveIt
- Based out of Boulder, CO

Q&A