PyConnect

A Lightweight Python-C++ Integration Framework

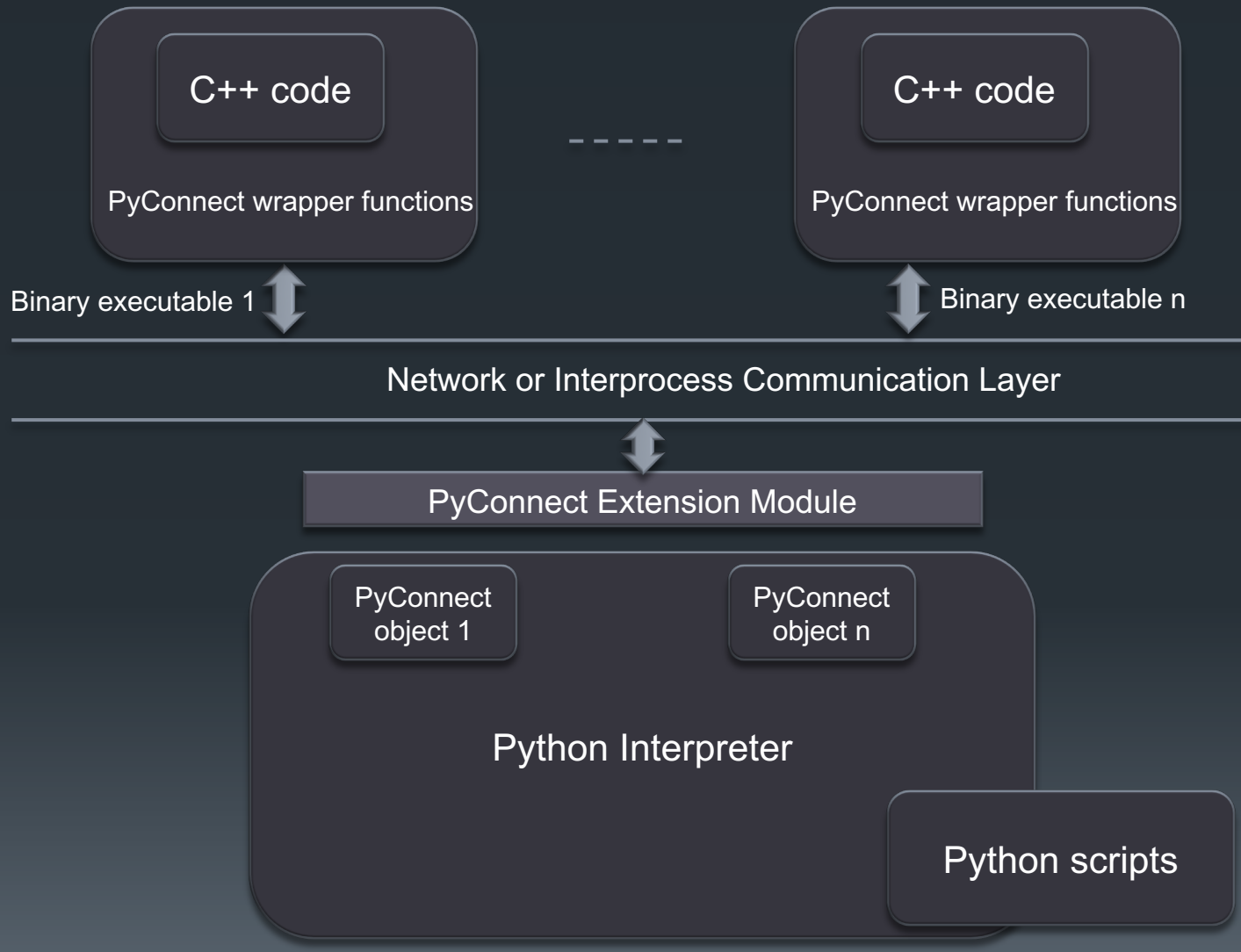
Overview

- Introduction.
- Architecture.
- Integration with Existing C++ Programs.
- Python Scripting with PyConnect.
- Limitation and Future Enhancements.

Introduction

- PyConnect provides a lightweight binding mechanism for C++ and Python.
- PyConnect provides a platform for integrating disparate C++ and Python program modules together with minimal efforts.
- PyConnect supports a distributed computing architecture with a star topology.
- PyConnect is written in portable C++ and supports multiple operating systems including Windows, OS X and other Unix like systems.
- PyConnect was originally developed for solving a problem of integrating existing Python Scripts and C++ code (developed by different parties) running on the Sony AIBO robot platform.

System Architecture



Architecture

- Client-Server architecture.
  - C++ programs are wrapped in “pythonised” executable objects with the use of PyConnect wrapper functions.
  - Functions and variables in the C++ programs are exposed using PyConnect wrapper macros.
  - PyConnect extension module loaded in a Python interpreter connects the “pythonised” C++ programs and creates corresponding PyConnect objects in the Python engine.
  - Python scripts access the functionalities of the “pythonised” C++ programs through their respective PyConnect objects in Python.
- Many-to-many relationship.
  - A PyConnect enabled Python engine can connect to many “pythonised” C++ programs running on the network.
  - A “pythonised” C++ program can support simultaneous connections from multiple PyConnect enabled Python engines on the network.
  - Multiple PyConnect enabled Python engines can communicate among themselves.

Architecture

- Event driven processing.
  - Creation, destruction and execution feedbacks of “pythonised” C++ programs are based on callback funtions.
- Auto-service discovery.
  - PyConnect uses an auto-discovery mechanism to discover services (provided by “pythonised” C++ programs) available on the Local Area Network(LAN).

Integration with Existing C++ Programs

- Prerequisites
  - The existing C++ code must have a top-level class that acts as a container for all functional methods and attributes of the program.
  - PyConnect will turn the program into a server program that permits simultaneous *stateless* access from multiple remote python engines. If the exposed functional methods are dependent on known states, addition code need to be added to the program to handle state information for different clients.

Integration with Existing C++ Programs

In your C++ program header file:

- Import PyConnect wrapper headers:

```
#include "PyConnectWrapper.h"
#include "PyConnectNetcomm.h"
```

- Define exposed "pythonised" module name

```
#define PYCONNECT_MODULE_NAME TestSample1
```

- Define a top-level class of the module:

```
class TestSample1:public OObject
{
public:
  TestSample1();
  ~TestSample1();
  void helloWorld();
  void printThisText( const std::string & text );
//…
};
```

Integration with Existing C++ Programs

- Declare PyConnect wrapper and communication layer:

```
PYCONNECT_NETCOMM_DECLARE;
PYCONNECT_WRAPPER_DECLARE;
```

- Set module description:

```
PYCONNECT_MODULE_DESCRIPTION( "A simple test program
that uses PyConnect framework." );
```

- Declare public methods to be exposed:

```
PYCONNECT_METHOD( helloWorld );
PYCONNECT_METHOD( printThisText );
PYCONNECT_METHOD( doMultiply );
```

Integration with Existing C++ Programs

- Declare public variable to be exposed:

```
PYCONNECT_RO_ATTRIBUTE( methodCalls ); OR
PYCONNECT_RW_ATTRIBUTE( methodCalls );
```

- Check `test_sample1.hpp` file in the testing directory for the complete code example.

Integration with Existing C++ Programs

In your C++ program cpp source file:

- In the module class constructor define details of exposed module, methods and variables

```
EXPORT_PYCONNECT_MODULE( TestSample1, "A simple
test program that uses PyConnect framework." );

EXPORT_PYCONNECT_RO_ATTRIBUTE( methodCalls );

EXPORT_PYCONNECT_METHOD( helloWorld );

EXPORT_PYCONNECT_METHOD( doAddition );
```

Integration with Existing C++ Programs

- Initialise the “pythonised” module and enable data communication.

```
PYCONNECT_NETCOMM_INIT;

PYCONNECT_NETCOMM_ENABLE_IPC; // for local
 interprocess communication

PYCONNECT_NETCOMM_ENABLE_NET; // for network
 communication

PYCONNECT_MODULE_INIT;
```

- Note that, `PYCONNECT_NETCOMM_ENABLE_IPC` is not defined on the windows platform.
- In the destructor of the module class

```
PYCONNECT_MODULE_FINI;

PYCONNECT_NETCOMM_FINI;
```

Integration with Existing C++ Programs

- Modification in program main execution loop:
  - If the program does not have a predefine main loop, in the `main()` function after the module class being instantiated add following:

    ```
    PYCONNECT_NETCOMM_PROCESS_DATA;
    ```

    See `test_sample1.cpp` for details.
  - If the program has pre-existing main loop:
    - Modify the module class to inherit from `FDSetOwner` class and implement the abstract functions defined in `FDSetOwner` class.
    - Add `-DHAS_OWN_MAIN_LOOP` flag as a compiler option.
    - Add `PYCONNECT_EXTCOMM_PROCESS_DATA( fd_set );` in the main loop.
    - See `test_sample2.hpp`, `test_sample2.cpp` and `CMakeLists.txt` for example.

Integration with Existing C++ Programs

- Push variable value updates to Python engine.
  - When the value of an exposed variable is modified, the updated value does not get to push to PyConnect enabled Python engine automatically. You need to add the following after code that modifies the variable value where it is appropriated.

    ```
    PYCONNECT_ATTRIBUTE_UPDATE( variable_name );
    ```

  - Note: do not use this macro too zealously. You need to consider communication cost it incurs.

Integration with Existing C++ Programs

- PyConnect logging facility
  - You need to enable PyConnect logging in your code.
  - Declare PyConnect log file at the begging of program source code:

```
PYCONNECT_LOGGING_DECLARE( "testing.log" );
```

  - Initialise and finalise the logging facility at the beginning and at the end of main() function with the following:

```
PYCONNECT_LOGGING_INIT;
PYCONNECT_LOGGING_FINI;
```

  - Use the following log message functions to save message in the log

```
INFO_MSG( “message\n” );
DEBUG_MSG( “message\n” );
ERROR_MSG( “message\n” );
WARNING_MSG( “message\n” );
```

  - To disable log, define `RELEASE` flag in the Makefile/CMakeLists.txt.

Integration with Existing C++ Programs

- Makefile/CMakeLists.txt modifications
  - Add the path where PyConnect headers are located to your program `$INCLUDE path`
  - Do either
    - Add following PyConnect framework source code to your program for compilation:

      `PyConnectCommon.cpp`

      `PyConnectObjComm.cpp`

      `PyConnectWrapper.cpp`

      `PyConnectNetcomm.cpp`
    - Compile pyconnect wrapper library using Cmake and link library: libpyconnect_wrapper.a into the program.

Python Scripting with PyConnect

- Build PyConnect extension module (command line)
  - Building

    ```
    python pyconnect_ext_setup.py build
    ```
  - Installing

    ```
    python pyconnect_ext_setup.py install
    ```
  - See README.md for details.

Python Scripting with PyConnect

- Enable PyConnect extension module in a Python engine:

  ```
  import PyConnect
  ```

- Discover existing “Pythonised” executables (PyConnectObject) on the network:

  ```
  PyConnect.discover()
  ```

- When a “Pythonised” executable connects to (or disconnects from) the PyConnect enabled Python engine, the following Python callback functions are executed respectively:

  ```
  PyConnect.onModuleCreate
  PyConnect.onModuleDestroyed
  ```

  - Implement matching functions and assigned them as the callback functions.
  - See `test_script.py` for details.

Python Scripting with PyConnect

- Communications between PyConnect enabled Python engine with “Pythonised” C++ executable objects are asynchronous.
- Variable value updates and returning values of method calls are notified to Python scripts through various callback functions on the corresponding *PyConnectObject*. Below is the general syntax for the callback functions:
  - For setting a new value for a variable:

    ```
    PyConnectObject.onSet[VariableName]( new_value )
    PyConnectObject.onSet[VariableName]Failed( error_code )
    ```

  - For variable value update:

    ```
    PyConnectObject.on[VariableName]Update( new_value )
    ```

  - For calling a method:

    ```
    PyConnectObject.on[MethodName]Completed( return_value )
    PyConnectObject.on[MethodName]Failed( error_code )
    ```

- Implement matching functions and as them as the callback functions
  - For example `pTS2.ontimerTriggerNoUpdate = onTimer`

Limitations

- PyConnect wrapper only supports following generic C++ data types:
  - Boolean, integer, string, float, double and void.
  - More complex data type/structure would additional data encoding, e.g. JSON format, into string.