Proposal for Robos, a Platform Independent Robot Operating System

Andrew Wood

EECS 395 Senior Project

Case Western Reserve University

Abstract

The purpose of Robos is to provide a scalable, fast, and computationally powerful development platform that is easy to use and provides versatile programming options for developers. Robos acts as an operating system that provides an environment where developers include programs as “plugins” that can be easily added, removed, and modified without having to change the rest of the system. Robos will be used to develop software that governs the operation of any robotic system, or generally any system that requires information flow.

Robos will have an application programming interface (API) that provides powerful computational tools and defines how client programs are executed. Clients will develop code that will be preferably built using a provided build system, but can be built in any manner as long as Robos is included. Robos will accept client defined programs and client defined data types, and will execute them in client implied order. The API is configured to handle most of the difficult challenges of developing concurrent, performance based code, and is designed to take most of the load of programming difficulty off of the developer, so that the developer can focus instead on the tasks their code is designed to solve. Robos is designed to fail gracefully and quickly, and is written to catch as many errors as possible at compile time. Robos tries to handle as much if not all of the concurrency in the client system as possible, but provides support for client concurrent problems.

Table of Contents

Abstract

Table of Contents

1. [Introduction](#Propsal_Introduction)
   1. [Relevant Background](#Propsal_Relevant_Background)
   2. [Project Goals](#Propsal_Project_Goals)
   3. [Related and/or Similar Systems](#Propsal_Related_Systems)
   4. [Project Advantages](#Propsal_Project_Advantages)
   5. [Project Disadvantages](#Propsal_Project_Disadvantages)
2. [Application Description](#Propsal_App_Discription)

2.1 [Project Constraints](#Propsal_Constraints)

2.2 [Project Challenges](#Propsal_Challenge)

2.3 [Proposed Project Solutions](#Propsal_Solutions)

1. [Software Requirement Specifications](#Propsal_Software_Requirements)
2. [Project Management Plan](#Propsal_Plan)

1 Introduction

* 1. Relevant Background

When developing complicated software systems, compartmentalization is an important guideline for producing quality software. When exercised, software systems that follow compartmentalization are easy to maintain, understand, and modify. In the world of robotics, typical robotic software packages are comprised of hundreds if not thousands of different programs that perform distinct tasks. There still is an issue with the amount of encapsulation in such a system; renaming or replacing a program requires modifying all dependent programs to reflect the change.

A more abstract version of compartmentalization involves abstracting away the interactions between programs. In simple terms, Robos aims to arbitrate the interactions between programs in complicated software systems, and aims to provide a safe environment in which programs can execute.

Robos achieves this by requiring client programs communicate via Messages. A Message is a client defined collection of data that is unique to the client program’s output (for example a sensor program would output a Message full of measurement data, etc). Robos manages the Messages that are “published” by a client program. Messages have a “topic” that client programs can subscribe to. When a Message is “published,” Robos will then execute all client programs that “subscribe” to that particular Message’s “topic.” This strategy reduces the interfacing between client programs into the interface between a client program and Robos. A client program will only execute if a Message whose topic the client program “subscribes” to is “published.” This is true compartmentalization; client programs are dependent upon the data they need to operate on existing instead of other client programs themselves existing.

Robos executes client programs in a smart, efficient manner. Code can have a significant performance boost if executed concurrently, except for when concurrent code operates on the same data. In the case of shared data, concurrent programs may have to wait and access the data one at a time. Waiting can decrease the performance gain achieved by executing concurrently in the first place, so Robos is designed to execute asynchronously. Simply, asynchronous execution is concurrent execution where code is only executed concurrently when needed, and can be used to reduce if not eliminate most conditions where concurrent code will wait.

* 1. Project Goals
* User Interface Project Goals:
* To create an application programming interface (API) (Robos API) that guarantees client code execution in a concurrently and program semantically correct manner, and provides powerful computational tools.
* To create an API (Logging API) that allows clients to log messages in a concurrently safe manner to runtime defined locations.
* To create an API (Async API) that guarantees asynchronous execution in a safe manner that supports client defined types.
* To create an API (Utilities API) that provides concurrent data structures as well as concurrent programming primitive types.
* To develop a platform independent build system that is used by clients to configure and build client code with Robos.
* Internal Project Goals:
* To create platform independent, build configurable software packages that implement all mentioned APIs that maintain correctness and asymptotic runtime guarantees independent of platform.
* To organize project directory structure to incentivize inclusion of client projects into existing build system
* Stretch Goals:
* To develop an Artificial Intelligence framework that folds into existing APIs and software packages.
* To incorporate concurrency limits like specifying the maximum number of threads that can be used into Asynchronous Execution.
* To incorporate graceful client program error handling such as restarting client program on error and disabling malfunctioning client programs.
* To incorporate built in internet streaming capabilities to support remote usage of Robos.
* To develop a version of Robos that does not dynamically allocate for usage in extreme embedded environments.
* To develop security features to protect integrity of client code against malicious attacks.
  1. Related and/or Similar Systems

**ROS (Robot Operating System)**: ROS is an open source operating system that provides a safe execution environment for client programs as well as powerful computational tools for robotic development.

**Pros:** Very flexible, the standard for robotic software development. It is fully optimized for specific flavors of Unix distributions. Very powerful tools available.

**Cons:** Has a large memory footprint. Has a high learning curve to get adjusted to developing under ROS. Exists only for specific flavors of Unix distributions. Only supports C++ and Python development (Java version released a little while ago, unclear if Java version supports other languages as well). Only supports certain sensors. Documentation can be confusing for inexperienced developers.

* 1. Project Advantages

Robos provides the following advantages for robotic development:

* Powerful computational tools such as internal transform conversion, coordinate system synchronization. Robotic data structures.
* Client code independency: client programs can be modified, deleted and added as necessary with no change to other client programs.
* Platform independent development.
* Client language independency; Robos will support development in many different languages. The complete list of supported languages will be posted as support is added.
* Provided build system. Little overhead for clients to add their code to a fully functional build system.
* Platform independency; clients are not restricted to specific operating system and architecture pairs for development.
* Ease of use. Stems from platform independency and language independency.
* Asynchronous execution with guarantee for correct concurrent scheduling and execution.
* Small memory footprint; Robos is designed to be small.
  1. Project Disadvantages

Robos may be limited by the following disadvantages:

* Platform independency: Being independent may limit some aspects of Robos from achieving full optimization.
* Small memory footprint. In an effort to minimize the size of Robos, some large frameworks and libraries may be omitted and a reliance upon the C++ standard library implementations.
* Client language independency: The interface between different languages could pose a performance bottleneck that would not exist if clients were restricted to developing in C++.
* Client build system. Pre-built binaries will not be available, clients will have to build Robos on the target platform.
* Client code will have to be “wrapped” in a C++ class. This class acts as the interface between Robos and the client code, and will also act as the interface between C++ and the client code language.

1. Application Description
   1. Project Constraints

Robos is constrained by its architecture. Robos is designed to be built either as a static or shared library (configurable via the client build system), and then linked to client code during the client build process. Since Robos manages client defined Message types that inherit from a base type that Robos exposes, client types are not known during the Robos build. Robos has no choice but to deal internally with the parent Message type, so when Robos is invoking client code with a Message, it is up to the Client to determine the child type of the Message and invoke the correct functionality. This can be a slight performance bottleneck due to multiple casting of Message instances.

Robos is also constrained by the promise of platform independency. There are serious optimizations that are missed by omitting platform dependent code. However, a partial solution to this is to wrap platform dependent code into Robos files that are expanded upon a build of Robos. The downside to this solution is an increase in compile time, and it still does not allow all platform dependent code to be included.

Robos is constrained by the build system. For clients to become familiar with the build system, and how to develop with the build system requires documentation and tutorials. This means clients will have to read the documentation and follow the tutorials before developing. However, this solution is still better than the level of complexity required to become familiar with ROS.

* 1. Project Challenges

**1 Accuracy:** In order for Robos to be useful, it must guarantee that client code is executed in a concurrently correct manner and follows the order of execution that Message types subscription and publication imply.

**2 Speed:** Robos must execute quickly between Message publication and client program invocation indicated by Message subscription. Robos must also effectively manage concurrency, as too many threads will induce thrashing and a severe loss in performance. In addition, utility functions and data structures must be fast with respect to client code execution in order to be useful.

**3 Scalability:** Robos must be able to support robotic software systems compromising of hundreds if not thousands of client programs.

**4 Platform Independency:** Robos must be able to be built and support client code on all platforms introduced to it.

**5 Client Language Independency:** In order for Robos to be useful to inexperienced developers, Robos must support a wide variety of languages so that developers are not constrained by language choices. This assumes that developers are aware of the potential performance decreases.

* 1. Proposed Project Solutions

**1 Accuracy:** Robos is built using the continuous integration approach. Robos is heavily tested as developed, and errors are addressed as they are encountered. Appropriate concurrent testing is used, so upon a successful build there is a high level of confidence on accuracy.

**2 Speed:** Robos itself is designed to be small. Robos mostly consists of implementation code, while the code that actually manages client programs is small. Asymptotic runtime guarantees are maintained and listed in the corresponding APIs. Robos can be thought of a conductor, while its internal libraries are thought of as sections of the orchestra. Robos is very small itself; its function is to exercise the internal libraries with client programs.

**3 Scalability:** Robos uses scalable data structures in the areas that will feel the effects of large inputs. Robos also tries to minimize each memory footprint as an attempt to take advantage of cache sizes for scalability and performance. Robos also omits large third party dependencies, assuming that clients will include large libraries at their own risk. Robos also manages a scalable concurrency model that can be client defined at the clients own risk.

**4 Platform Independency:** Robos tries to ignore platform dependent code and takes advantage of platform dependent features of C++. Robos also is supported by a platform independent build system.

**5 Client Language Independency:** Robos will conduct research into data transmission between different languages, and will define template classes and procedures for language support.

1. Software Requirements

The Software Requirements Specification is attached to the end of this document.

1. Project Management Plan, Version 1

At the time of this document, Logging, Utilities, and Async are all working in non-optimized forms. The Robos package itself is under development. The build system is fully functional minus code coverage, static analysis, and API generation:

|  |  |
| --- | --- |
| Task | Estimated Completion Date |
| Implement Client Lookup in Invocation in Robos | February 20, 2016 |
| Add Code Coverage to Build System | February 20, 2016 |
| Implement Logging in Async, Utilities | February 22, 2016 |
| Add Static Analysis to Build System | February 23, 2016 |
| Implement Client Registration in Robos | February 24, 2016 |
| Implement Time Stamping in Robos | February 24, 2016 |
| Add documentation generation to Build System | February 26, 2016 |
| Add killswitch for client execution in Robos | February 27, 2016 |
| Write Robos\_FT (functional test client) | March 1, 2016 |
| Add Transform data structures, Quaternions, Coordinate System to Utilities | March, 10, 2016 |
| Generate final documentation and finalize Test Suites (Unit Tests, Functional Tests, Regression Tests) | March, 15, 2016 |
| Begin Case Robotics Club Integration | March 19, 2016 |
| Develop analytics for performance in real usage environment | March 28, 2016 |
| **Stretch Goals** (AI incorporation, advanced concurrency model, language expansion, security hardening, internet streaming, etc.) | April 1, 2016 |
| Submit Final Project Presentation  (all deliverables with test client and potentially Case Robotics Club usage) | April 26, 2016 |

|  |  |
| --- | --- |
| Milestones | Estimated Completion Date |
| Allow clients to interact with Robos | March 2, 2016 |
| Complete working proof of concept including test client | March 16, 2016 |
| Publish final proof of concept API | March 16, 2016 |
| Submit Final Project Report and Code | April 20, 2016 |
| Release Robos to Case community for feedback | April 29, 2016 |

Robos

Software Requirements Specification

Version 1.0

February 19, 2016

Prepared by Andrew Wood

EECS 395 Senior Project

Case Western Reserve University

Table of Contents

Table of Contents

Revision History

1. [Introduction](#SRS_Introduction)
   1. [Purpose](#SRS_Purpose)
   2. [Scope](#SRS_Scope)
   3. [Document Conventions](#SRS_Conventions)
2. [Overall Description](#SRS_Description)
   1. [Product Perspective](#SRS_Perspective)
   2. [Product Functions](#SRS_P_Func)
      1. [Necessary Functions](#SRS_N_Func)
      2. [Additional Functions](#SRS_A_Func)
   3. [User Classes and Characteristics](#SRS_User_Classes)
   4. [Operating Environment](#SRS_Operating_Env)
   5. [Design and Implementation Constraints](#SRS_Constraints)
   6. [User Documentation](#SRS_User_Doc)
   7. [Assumptions and Dependencies](#SRS_Assump)
3. [External Interface Requirements](#SRS_External_Interface)
   1. [User Interfaces](#SRS_UI)
   2. [Hardware Interfaces](#SRS_HI)
   3. [Software Interfaces](#SRS_SI)
4. [Functional Requirements](#SRS_Func_Req)
   1. [Async](#SRS_Func_Async)
      1. [Making Asynchronous Service Calls](#SRS_Func_Async_Calls)
      2. [Continuations](#SRS_Func_Async_Continu)
   2. [Logging](#SRS_Func_Logging)
      1. [Logging Output](#SRS_Func_Logging_Output)
   3. [Utilities](#SRS_Func_Utilities)
      1. [Using Concurrency Primitives](#SRS_Func_Utilities_Prim)
      2. [Accessing Data Structures](#SRS_Func_Utilities_DS)
   4. [Robos](#SRS_Func_Robos)
      1. [Defining Message Types](#SRS_Func_Robos_MSG)
      2. [Wrapping Client Programs in C++ Nodes](#SRS_Func_Robos_Wrapping)
      3. [Subscribing Nodes with Robos](#SRS_Func_Robos_Sub)
      4. [Writing Callbacks](#SRS_Func_Robos_Callbacks)
5. [Non-functional Requirements](#SRS_Non_Func)
   1. [Performance Requirements](#SRS_Perf_Req)
   2. [Security Requirements](#SRS_Sec_Req)
   3. [Software Quality Attributes](#SRS_Soft_Attr)

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Date | Change Reason | Version |
| Andrew Wood | 02/11/2016 | Project Proposal Submission | 1.0 |

1 Introduction

* 1. Purpose

The purpose of this document is to specify the functional and nonfunctional requirements supported by version 1.0 of the Robos Operating System. Robos will consist of several libraries; Logging, Utilities, Async, and Robos, with each supporting its own API. Robos will also include a platform independent build system. This document is to be used to specify and compare the implantation requirements of Robos, and will be included in its version 1.0 release unless otherwise specified.

* 1. Scope

Robos will allow clients to develop software to be used in a robot or other informationally dependent system. Robos will provide a safe and extremely compartmentalized environment in which to develop in and abstract away the difficulties of higher level programming in order to help developers focus on the semantics of their code. The scope of this project will be constrained to designing, implementing, and testing both the client side of developing using Robos and the execution environment with which Robos provides. Robos will be designed to manage all client code in a quick, efficient manner while utilizing the smallest memory footprint available and provide powerful computational tools.

* 1. Document Conventions

This document will use the term “Robos” to refer to the general project as a whole, and also to refer to the library named “Robos” that provides the user interface. This document will also interchange the terms “clients” and “developers” to refer to users developing code that sits atop Robos. The term “node” will be used to represent the C++ wrapper class for client programs used in Robos.

1. Overall Description
   1. Product Perspective

Robos is designed to efficiently execute client programs designed to manage a robot. The Operating System will consist of several subprojects linked together to achieve efficient execution. Robos will be dependent upon client code for execution, otherwise it will have nothing to manage, and nothing to do. Robos execution is heavily dependent upon the asynchronous execution library developed, and the typing system involved to direct information flow in client implied order. A diagram of how client programs are executed in Robos is below:

Robos

Node Database

Scheduler M

Scheduler B

Scheduler A

Master Node

Where black arrows are Messages, red arrows are Node instances, and blue arrows are asynchronous work by invoking a Node with the appropriate message.

* 1. Product Functions

Below is a list of functions that are necessary for Robos to have the functionality described. Below that are Additional Functions, which are stretch goals and may be unimplemented due to time constraints. The function input and output is described in section 4 of this document.

* + 1. Necessary Functions

Below are the necessary functions broken up by component:

Async

Making Asynchronous Service Calls

* Makes Promise of eventual function fulfillment.
* Allows more efficient concurrent execution.

Continutations

* Chain asynchronous events together.
* Sequential execution of logic.
* Execute different branches of logic depending on result of asynchronous execution.

Logging

Logging Output

* Helps debugging across different components executing simultaneously.
* Redirect output to generic file directories.

Utilities

Using Concurrency Primitives

* Semaphores implemented.
* Reduces complexity of concurrent programming than using standard C++ concurrency primitives.
* Adoptability of Semaphores into standard C++ concurrency primitives.

Accessing Data Structures

* Exposing concurrent and scalable data structures for client and internal usage.

Robos

Defining Message Types

* Client defined Message types are passed between client programs.
* Messages can be arbitrarily sized but only contain primitive types (Strings are included).
* Has a “topic” that summarizes what data a Message contains.

Wrapping Client Programs in C++ Nodes

* Nodes are C++ classes that invoke client programs with Messages and are the interface between the client program and Robos.
* Represents how client programs are stored and executed internal to Robos.

Subscribing Nodes with Robos

* Register Node with Robos.
* Can be invoked after registration (assuming no duplicate node exists).

Writing Callbacks

* Describes how a Node is invoked.
* Client logic to invoke the correct client program functionality given a Message. Analogous to the “main” function given runtime arguments.
  + 1. Additional Functions
* Security
* Give Nodes security constraints just like users in an Operating System.
* Harden internal functions against standard attacking strategies.
* Internet Streaming Capabilities
* Stream statistics in real time.
* Receive input commands for additional client control.
* Use remote procedure calls to take advantage of cloud computing for expensive operations.
* Artificial Intelligence Utilities
* State space searching algorithms.
* Machine learning algorithms.
* Data structures for AI expansions.
* Concurrency Configurations
* Specify maximum number of threads.
* Distribute threads according to execution scheme (based on the clients and Message types in the system).
* Client Error Handling
* Restart client programs if malfunctioning or crash system if error in critical system.
* Stream error diagnostics, log error diagnostics locally.
  1. User Classes and Characteristics

Robos exposes two classes for clients to adopt. They are listed below as well as their characteristics:

1. NodeBase

* Client programs must be “wrapped” in a C++ class that represents the interface between the client program and Robos. That class represents a node internally to Robos, and must derive from NodeBase.
* NodeBase has one method that must be overridden which is invoked when Robos schedules that node for execution. It is left for the client to implement as that client program may perform different functionality with different Message types.

1. MessageBase

* Client programs interact with each other by passing Messages. A Message is a client defined data type that must derive from MessageBase. MessageBase is the way Robos stores a Message internally and is what is passed to a client program upon execution.
* Client programs subscribe to a Message topic, which is a public member of MessageBase. This value is set upon construction of the Message.

There are no other classes that must be adopted by clients. Robos exposes a class called “Robos” but that is non-derivable and acts as a singleton into Robos.

* 1. Operating Environment

Robos is designed to be platform independent. It can run on Windows, Unix and OSX machines, with different Operating Systems being tested each week. It is designed to operate in embedded environments, and can operate on standard laptops and desktops as necessary. The memory used in the dynamically allocating version grows in proportion to the number of client programs running. There is no memory estimate on the non-dynamically allocating version currently.

* 1. Design and Implementation Constraints

The main design constraint will be language support. There is a significant performance bottleneck with interfacing C++ and another language. This is an assumed risk, as the choice of development language is left to the client, so is a risk that Robos is designed to handle, but not removed from functionality.

* 1. User Documentation

Robos will be built on each development platform, and documentation will be generated as part of the initial build. If unsatisfied or upon failure, the API will be available through Google search. As a last resort, clients will have access to the source code and all developer documentation included with the build system.

* 1. Assumptions and Dependencies

Robos is dependent upon the skill of the client. It is a useful system based on the assumption that clients are either experienced and develop quality code, or are willing to read the documentation and learn how to develop with Robos. There is no software block for client code that does not terminate and will follow client defined procedures both in the concurrency model, and in the code that is executed.

To build Robos, the following dependencies are required:

* Python 2.7.10 or higher.
* CMake 3.3 or higher.
* A C++ compiler that supports C++11.

1. External Interface Requirements
   1. User Interfaces

The user interface of Robos is the build system. Client code is built and linked to the Robos library, producing an executable that, when run, executes the system. If implemented, the built in internet streaming functionality will provide a user interface during runtime.

* 1. Hardware Interfaces

Robos interfaces with sensors installed on a robot that Robos is running on. Robos interfaces with the sensors through client code, as sensor drivers are a client responsibility.

* 1. Software Interfaces

Robos interacts with client software as a framework. In other words, Robos calls into client code, while client code can call into Robos to invoke specific functionality, as in publishing a Message or invoking a built in function or using built in types. Robos interacts with other software primarily through the Messaging and Node based data types, which clients derive from.

1. Functional Requirements

Below are the functions specified in section 2:

* 1. Async
     1. Making Asynchronous Service Calls

**Description:** Async allows C++ function pointers to be executed asynchronously on worker threads. Upon instantiation, worker threads are spawned according to a predefined scheme at a well known location in the build system. std::function objects can then be submitted for execution on one of these threads. Threads are organized by job type with the intent that a std::function is given to the worker thread whose job it is to execute code of a given intent.

**Use:** Steps:

1. Link client code with Async library.
2. Include AsyncExecution.hpp in client code.
3. Make a procedure call to Async::Execute(std::function, std::string) where the std::function is the code to execute and std::string is the name of the Scheduler that will schedule and execute the provided code.
   * 1. Continuations

**Description:** An Asynchronous Service Call returns a Promise object. A Promise object represents the “Promise” that is being made to either return the output of the asynchronous procedure or the error that was thrown. A Promise can be given any number of std::function objects that it will schedule for execution upon the settlement of the Promise. This is used to execute sequential blocks of logic.

**Use:** Steps:

1. Make an asynchronous service call and get the returned Promise object.
2. Make a call to <Promise\_object>->Then(std::function, bool) where the std::function is the code to be executed on settlement and the bool is true if the std::function is to be executed upon a successful settlement of the Promise (there was no error in the original asynchronous service call client code).
   1. Logging
      1. Logging Output

**Description:** Logs messages to generic file locations. Files are created at runtime.

**Use:** Steps:

1. Link client code with Logging library.
2. Include ILogger.hpp and Factory.hpp files in client code.
3. Retrieve a logger instance using

Logging::Factory::makeLogger(<name>, <file\_path>)

1. Use any of the four functions:
2. LOG\_DEBUG(<logger>, <format\_string>, <msg\_args>…)
3. LOG\_ERROR(<logger>, <format\_string>, <msg\_args>…)
4. LOG\_WARNING(<logger>, <format\_string>, <msg\_args>…)
5. LOG\_INFO(<logger>, <format\_string>, <msg\_args>…)
   1. Utilities
      1. Using Concurrency Primitives

**Description:** Use Semaphores available in Utilities library.

**Use:** Steps:

1. Link client code with Utilities library
2. Include Semaphore.hpp.
   * 1. Accessing Data Structures

**Description:** Use data structures available in the Utilities library.

**Use:** Steps:

1. Link client code with Utilities library.
2. Include name of file to use.
   1. Robos
      1. Defining Message Types

**Description:** Client defined Message types are how information is passed between client programs. These Messages contain all relevant data for downstream processing and have a topic that summarizes the type of Message. Messages can only have primitive data types, but strings are an exception to this case.

**Use:** Steps:

1. Link client code with Robos library.
2. Include MessageBase.hpp.
3. Derive child structure from MessageBase.
   * 1. Wrapping Client Programs in C++ Nodes

**Description:** Client programs need to interface with Robos. Robos also needs to store client programs internally and invoke them with Messages. Therefore, all client programs need to be wrapped into C++ node classes for Robos to interface with.

**Use:** Steps:

1. Link client code to Robos library.
2. Include NodeBase.hpp.
3. Derive child structure from NodeBase.
4. Implement “MainCallback” method. This method takes a MessageBase type and produces a MessageBase type. It is left up to the client to determine what Message types trigger what client functionality.
   * 1. Subscribing Nodes with Robos

**Description:** Nodes are invoked with Messages, which in turn pass the Messages on to the client programs that the nodes wrap. For Robos to invoke a Node with a Message, Robos must know what types of information a Node wants to receive. Upon creation of a Node, a Node must make a call to the Robos::Register(<Node>, <Subscription\_topics>…) function, which tells Robos to invoke <Node> if a Message is published that has a topic that exists in <Subscription\_topics>…

**Use:** Steps:

1. Link client code to Robos library.
2. Derive client structure from NodeBase following procedure above.
3. In constructor of child structure, make a procedure call to Robos::Register(…) function, or in another file that knows a client structure instance has been created, make a procedure call to Robos::Register(…) with the child structure instance.
   * 1. Callbacks

**Description:** Nodes are invoked with Messages, and the client decides what Message types invoke what client program functionality. When creating a Node type, the client must override the <NodeType>::MainCallback(Message) method. In this method, the Message must be casted down to its child type, and then passed to client program callback. When <NodeType>::MainCallback(Message) is invoked by Robos, the Message will propagate down to the correct callback to execute.

**Use:** Steps:

1. Follow Procedure 4.4.3 above.
2. When overriding <NodeBase>::MainCallback(Message) method, determine type of Message, cast it to the child type (known in client code), and pass the casted instance down to a client defined method.
3. Non-functional Requirements
   1. Performance Requirements

Robos is required to run quickly and efficiently in order to be a useful product. Most Robos functions are constant time operations, with the exception for Node invocation. This is due to the nature of having to search a database, which can increase the asymptotic runtime of this function. However, well tested and efficient usage of B+ tree structures limit this runtime to very small values for even large B+ tree structures. With this exception, Robos functions operate in constant time in an effort to execute client code as fast as possible. Utility functions run proportional to the input size given as most are data structures that operate on vectors or matrices of data.

Should AI algorithms be implemented, some algorithms have asymptotically expensive runtimes, and should be invoked with the asymptotic runtime in mind. Overall, the runtime of Robos is dependent upon the asymptotic runtime of client code. Robos can avoid slowdown with quick, clean code from clients.

* 1. Security Requirements

Robos initially will have no security features implemented. This only prevents a small risk because Robos does not initially support internet communication, so the only attack vector into a Robos system would come from a client Internet node or an attacker physically downloading attack code onto the hardware. These risks can be mitigated by the client.

If security features are implemented, they will focus around any internet connectivity functionality, as well as provide security privileges to client defined resources for all nodes in the system. A client will configure security privileges and develop the nodes to exercise the privileges.

* 1. Software Quality Attributes

The software of Robos shall be well organized and easily understandable. Accurate naming conventions will be used on all parts of the software packages, and descriptive, helpful, and accurate documentation will be provided along with the software. Additionally, a fully functioning build system will be distributed along with Robos code in an effort to accelerate client development with the Robos platform.