UNIVERSAL PLUG-N-PLAY SENSOR INTEGRATION FOR ADVANCED NAVIGATION

THESIS

Daniel L. Elsner, Captain, USAF

AFIT/GE/ENG/12-12

AFIT/GE/ENG/12-12

UNIVERSAL PLUG-N-PLAY SENSOR INTEGRATION FOR ADVANCED NAVIGATION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Insitute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Daniel L. Elsner, B.S.E.C.E., A.L.M.

Captain, USAF

March 2012

AFIT/GE/ENG/12-12

UNIVERSAL PLUG-N-PLAY SENSOR INTEGRATION FOR ADVANCED NAVIGATION

Daniel L. Elsner, B.S.E.C.E., A.L.M.
Captain, USAF

Approved:

//signed//	9 March 2012
Maj Kenneth A. Fisher, PhD (Chairman)	Date
//signed//	9 March 2012
John F. Raquet, PhD (Committee Member)	Date
//signed//	9 March 2012
Maj Jeffrey M. Hemmes, PhD (Committee Member)	Date

Abstract

Non-GPS navigation is a subject receiving much interest in the military and technological communities alike. GPS has the ability the provide very precise positioning, particular in the case of differential GPS. Unfortunately, GPS cannot always be relied upon for a particular application. Currently, a non-GPS positioning system requires several or more sensors to resolve positioning; often, these systems still fall short of a GPS-based system. Meanwhile, sensors have become more powerful, cheaper to produce, and therefore more accessible, allowing for the design and testing a multitude of possible non-GPS based positioning platforms. Being able to combine a wide range of these sensors has spawned bold and novel results in the field of robotics and autonomous vehicles, however, these advances have not come without cost. In particular, the influx of such a broad number of sensors has presented significant software challenges. Engineers and developers benefit tremendously from the ability to use a wide range of sensors, however, the software used to use them for a given application is often hastily implemented in order to quickly gather data and analyze results at the expense of software maintainability or reusability. Ultimately, a scalable platform allowing for sensors to be quickly added, removed, or replaced is desired and this serves as the primary motivation for this research effort. Specifically, the requirements of such a system are outlined and attempts are made to achieve them using two separate systems: one using middleware and the other using more traditional software design patterns. The end result is not so much a deliverable in terms of software, but more of a feasibility analysis comparing the two approaches.

”To my wife and family, who inspire and support me through all of life’s challenges.”

Acknowledgments

Above all I thank God for all he has blessed me with in life. I thank my wife and three sons for all their love, support, and encouragement and for their understanding and efforts when my academic priorities called. I would also like to express my deep gratitude and respect to those who I have learned from along the way, to include classmates, staff, and faculty. I particularly thank those instructors who have taken on the additional roles of serving as members of my thesis committee. Last but not least, I thank my advisor for being a teacher, mentor, and leader whom I could depend on for instruction, advice, and guidance throughout my time here at AFIT.

Daniel L. Elsner

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Universal Plug-n-Play Sensor Integration for Advanced Navigation

I Introduction

This section briefly introduces the thesis topic, to include the research motivation and problem definition. The middleware to be used for a good amount of the sensor testing is mentioned as it ties directly to related work previously attempted. Finally, the research goals are addressed along with an outline of how the rest of the paper will be presented.

1.1 Motivation

As military and civilian positioning systems have become increasingly dependent, or even exclusively so, upon GPS for data collection and real-time navigation, it is vital to have alternative capabilities to serve in place of or complementary to it. GPS presently affords superb positioning resolution, however, there are cases in which it is not a viable option. For instance, INS provides much greater accuracy for aircraft flight and landing. Also, there are several scenarios that would prevent GPS from even being available, such as being in an indoor environment. For these reasons, it is important to consider navigation through the lens of signals of opportunity (SOO), such as landmarks for vision-aided input, magnetic field mapping, etc. This is where other sensors incorporated into a full-up navigation suite can greatly enhance accuracy and dependability.

This area of study is not new and is currently employed in the Advanced Navigation Technology (ANT) center at the Air Force Institute of Technology (AFIT). The position solution takes into account all sensor data and is resolved using Kalman filtering. With the ability to track sensor status-of-health (SOH) as conditions change, real-time positioning algorithms using inline Kalman filtering becomes feasible. Notwithstanding, other logistical considerations must also be addressed for it to be practical. For example, the payload for an all-terrain, autonomous vehicle is likely quite high but power and other tradeoffs must be weighted toward achieving the most desirable results for a given application. Similarly, a person has strict limits on the amount of weight he can transport so the number and effectiveness of each sensor needs to be carefully considered in designing a non-GPS positioning system for such a platform. With such varying navigation systems and modes which they could be deployed, a modular, scalable PnP sensor environment represents a vital component in realizing their implementation [9].

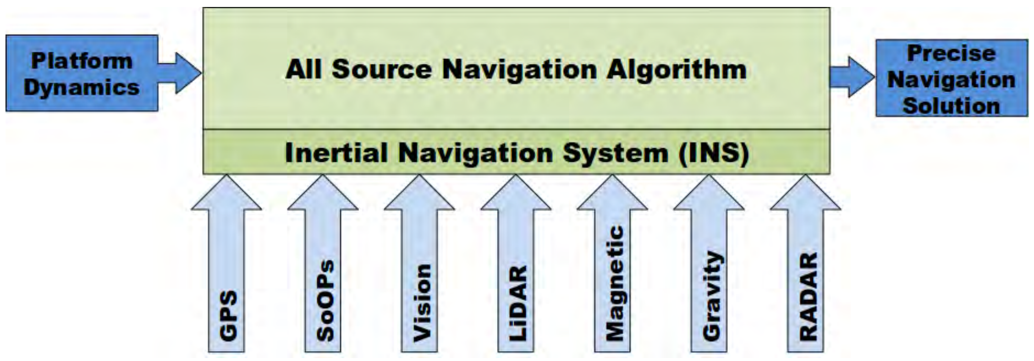

Platform Dynamics
All Source Navigation Algorithm
Precise Navigation Solution
Inertial Navigation System (INS)
GPS
SoOPs
Vision
LiDAR
Magnetic
Gravity
RADAR

Figure I.1: All Source Navigation Platform [9]

Figure I.1 shows a block diagram representing a platform consisting of multiple sensors. For an all-source navigation solution, the sensors listed capture a basic suite which is not necessarily all-inclusive. If the All-Source Navigation Algorithm (ASNA) is implemented on a PnP software environment, sensors can be added and replaced without disturbing the rest of the system. Perhaps more importantly, the algorithm will not require modification and the implementation won't require a restart. PnP sensor integration would allow ASNA to produce the best positioning solution based on the sensors available in real-time. In kind, it would do this for any configuration, be it one sensor strapped to a human being or multiple mounted to a robotics platform.

The use of robotics for Navigation and Positioning (N&P) is pervasive throughout industry, the military, and academia. The widespread availability of many different types of platforms and sensors provides a solution for nearly any desirable effect. Furthermore, the crossing of the robotics and PC domains allows programs to be easily implemented, tested, and monitored from virtually anywhere at any time. Because of this, along with the direct navigation and positioning robotics applications currently employed within the Department of Defense (DoD), a robotics test setup is a natural selection for the testing platform of this PnP experiment.

The advancement of robotics technology has been hindered significantly due to a lack of an integrating standard, such as what is currently enjoyed within the personal computing industry [16]. A PnP sensor solution could serve as a primary catalyst in obtaining this much needed change in robotics hardware and software architectures. This is quite desirable since a lack of clear standards has represented a major setback in the the robotics field [24]. Many robotics applications require custom hardware and software configurations to suit a specific need. With a pre-accepted architecture, this is not much of a problem. Since these configurations revolve primarily around sensor integration, a PnP sensor environment could be extended directly into an accepted architecture amongst sensor manufacturers, programmers, and robot users alike.

1.2 Problem Definition

The problem intended to be solved as a result of this research, in short, is the current inability within the ANT center to perform navigation and positioning experiments that are adaptive and reactive to the inputs of a range of various navigation sensors. It is desirable to have a testing platform with software capable of executing real-time navigation and positioning algorithms, responding to the data received from the sensors and selecting which to use based on various scenarios and metrics. For example, a vehicle using GPS to navigate autonomously should be able to change sensor configurations on-the-fly in order to navigate in an area not covered by GPS, such as an indoors. Currently, test runs are conducted and data is post-processed after they are completed. Ideally, with a system capable of real-time PnP sensor integration, a test vehicle could execute adaptive Kalman Filtering and other such algorithms as the environment changes in a fully autonomous manner. Having a system that is capable of recognizing, configuring, running and handling the data provided by any sensor added to it with the need to shut it down and start back up again would represent a mark a major step toward achieving this.

A secondary objective in obtaining a PnP navigation system is achieving a system that is highly scalable and incorporates elements of software reusability. Doing this helps to ensure enhancements can be performed in the future with as little pain as possible. The software itself should be capable of handling new sensors when they are connected, gather data from them, and pass that data along to whatever application needs to use it. The benefit of this is it allows positioning to be resolved faster and with a higher degree of precision by allowing real-time objective sensor configuration changes. Simply put, sensors can be added, removed, or replaced on-the-fly allowing their data can be incorporated as inputs to the algorithm with minimal delay.

This type of effort does not appear to have been completed in any institution or venue. However, the subject is one that AFIT, specifically the ANT center, has been considering for quite some time since many of its navigation and positioning tests involve multiple sensors. Defense Advanced Research Projects Agency (DARPA) has put out a proposal to address a very similar platform in their Broad Agency Announcement (BAA) for an All Source Positioning and Navigation (ASPN) solution [7].

PnP has seen a great deal of attention since the emergence of personal computing into nearly every household in the world. With the preponderance of electronic devices interfacing with computer systems using the Universal Serial Bus (USB) interface, users have become accustomed to plugging in components and having them ”just work.” The PnP specification was developed jointly by Microsoft and Intel to satisfy personal computing needs and began to see widespread use in 1995, coinciding with the release of Windows 95 [20]. The utility presented by this concept has since transcended into other realms of electronic technology, from entertainment systems to satellite buses. For this reason, the definition of PnP is usually extended to encompass a broader context: the ability to add a new component to a system and have it work automatically; without having to do any technical analysis or manual configuration [32]. This is the definition that will be used for the purpose of this paper.

1.3 Research Goals

The goal of this research effort is to specify, test and implement a plug-n-play (PnP) sensor integration environment for AFIT's ANT center. It will explore whether accepted software design patterns are adequate for building configurable robot control systems relying upon many common sensor devices for navigation. This effort will be heavily software oriented and will address the specification and implementation of various hardware recognition and communication algorithms spanning multiple system layers. This effort is driven by the ANT centers need for a better way to add and remove sensors from a particular network configuration. The current system does not support this, requiring specialized treatment of each sensor based on a particular application.

The main goal of this research is to satisfy the needs of the ANT center while being able to apply the knowledge and experience within the realm of software engineering AFIT has provided. Also, helping to enhance and promote code reusability, and apply this in practice, is a secondary goal since it creates a foundation upon which others may be able to build upon in the future.

In meeting the primary goal, there are several sub-level goals that need to be met. The first of these is to establish a manner in which to discover sensors when they have been connected to a system. After that, data needs to be obtained from the sensor and passed on to a user in real-time. There are several ways in which these effects may be realized, so the final goal would be to compare various methods and consider the implications they hold in obtaining future advancements in the same field of study.

1.4 Summary

The rest of this thesis will be organized according to the AFIT thesis style guide. The next chapter will introduce some of the key concepts pertaining this topic along with other related work that has been done. Chapter three describes the methodology used for the research and testing. Chapter four lists the results of the tests conducted and describes what worked and what didn't. Finally, the last chapter summarizes the entire research project, stating assumptions and conclusions drawn from the results. Furthermore, this chapter lists items left outstanding that could be useful to revisit later in a future research or follow-on effort.

II Background and Related Work

The current study will focus primarily upon test systems using robots configured with some of the most common navigation and control sensors. The subject of PnP is very much a hot topic in multiple electronics disciplines and much literature exists detailing how this may be possible in many different technological applications. Since this research focuses on PnP for sensor integration, it should be no surprise that this section will be limited to work that relates, either directly or indirectly, to this subject. Further, since the work depends upon work that is currently employed within the ANT center, aspects of navigation, positioning, and control will be highlighted, and this introduction will be expanded upon in later sections in terms of how it ties into sensor PnP. In particular, algorithms and experiments conducted on robotics platforms using Kalman filtering and Simultaneous Localization and Mapping (SLAM) will be recurring themes centered within the scope of this research study.

The actual methodology discussed in Chapter 3 uses many of these concepts and results, however, the list is not all-inclusive. In fact, some of the work presented in this section serves as a good reference for introducing key concepts and possibilities surrounding the technology, but is not used explicitly within this particular research. The experimentation will involve the use of standards introduced by the Object Management Group (OMG) for modeling: the Unified Modeling Language (UML), component-based software patterns, self-describing object behavior, and Robot Operating System (ROS) as the framework for the robotics implementation. The overall goal is eventual universality in terms of a given sensor setup, meaning that any sensor can be added to or removed from the system at any given time without additional setup requirements placed on the user. However, being that navigation sensors do not currently subscribe to a standard requiring self-describing behavior, for this to be realized would require a fundamental change in the current field which has yet to occur. Therefore, only a few sensors will be used as a proof-of-concept and will demonstrate how this can be eventually expanded into a universal solution.

The sections of this chapter are arranged logically, building up to the next section in experimental methodology. The first topic explored is component-driven design, UML, and various software paradigms and algorithms. The next section introduces various PnP work that has been done in the electronics field. Finally, ROS and other software development environments are introduced, demonstrating how they have been used to achieve results and solutions to similar problems.

2.1 Component-Driven Design

There are countless stories and real-life examples of large, monolithic software written with no overarching logical structure to speak of. This leaves code to be unreadable to virtually anyone except for, and sometimes including, the developer himself. For an organization, the cost of obtaining or even maintaining this specialized software has been quite resource intensive. This has led to a shift in traditional development to one which can benefit from the use of commercial-off-the-shelf (COTS) chunks, which can be logically labeled as objects or components. Since many of these terms will be used quite frequently throughout the rest of the text, it's important to first introduce these terms and explain what is meant by them.

2.1.1 Object-Oriented Programming. Object-Oriented Programming (OOP) began in response to the limitations of traditional, sequential type programming methods. In the traditional, sequential style of programming, the actions of a particular program, or logic, are of central focus. In OOP, a program is separated into objects, which are blocks of code serving as actors. One of the better descriptions I've found echoes this differentiation, stating that OOP focuses more on objects than actions; more on "data" than "logic" [39]. Oracle defines OOP as "a method of programming based on a hierarchy of classes, and well-defined and cooperating objects." The organization further explains objects as executable copies of classes, or class instances [29]. An object is very similar to a class, with a class having one or multiple objects. Not only does a particular object inherit methods and attributes from its parent class, it can also override them, or contain additional ones [44]. Perhaps the biggest advantage of using OOP is that objects can be written in isolation from the rest of the program providing they conform to the user-defined interface. This provides modularity since it allows multiple portions of code to be worked on concurrently and allows changes to be made without interfering with the overall functionality of a particular piece of software. Although these latter descriptions may appear a bit more cryptic, they introduce important concepts that will be used and described further in the text to follow.

2.1.2 UML. UML was developed by the OMG as a tool for modeling software, either whole or in part. According to OMG, UML is "the way the world models not only application structure, behavior, and architecture, but also business process and data structure" [12]. It "is a visual language for specifying, constructing, and documenting the artifacts of systems" [23]. There are many different types of UML artifacts, however, this study will only use a few primarily: 1) Use Cases, 2) Class Diagrams, and 3) Activity Diagrams. A brief description and graphic of each follows, with the figures provided as examples from within Enterprise Architect (EA).

Use cases are the primary way to describe actors and how they interact with a software system. The diagram is typically included along with a text description of each use case, depicting main success scenarios and alternative scenarios. A use case diagram will capture each use case for a given system. Use cases can be written in three formats: terse (short form), casual and fully dressed (long form) [23]. Only terse and fully dressed will be used in this study. An example use case diagram follows, where client and administrator are actors and the round circles represent the use cases.

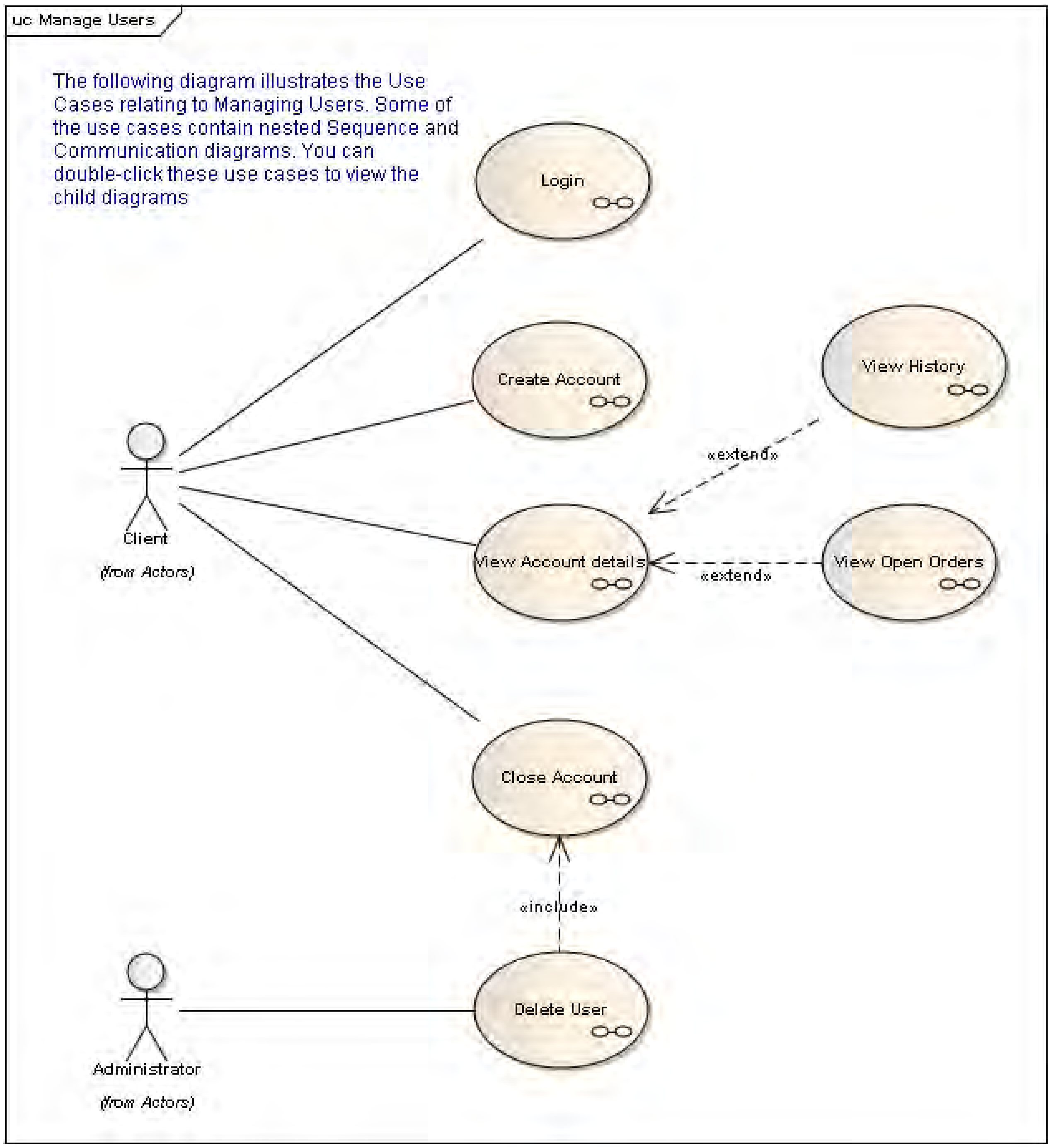

uc Manage Users
The following diagram illustrates the Use Cases relating to Managing Users. Some of the use cases contain nested Sequence and Communication diagrams. You can double-click these use cases to view the child diagrams
Login
Create Account
View History
«extend»
View Account details
View Open Orders
«extend»
Client
(from Actors)
Close Account
«include»
Delete User
Administrator
(from Actors)

Figure II.1: Example Use Case Diagram: Managing Users

An activity diagram is another way in which to model the use cases graphically, showing how activities flow between different actors within a system. They can be generated automatically using tools such as EA directly from the use case diagram. An example of an activity diagram is shown in Figure II.2. This particular example was taken from the Larman textbook because it provides additional details not included in the EA examples. Each column, known as "swim lanes," represent the control of different actors [23].

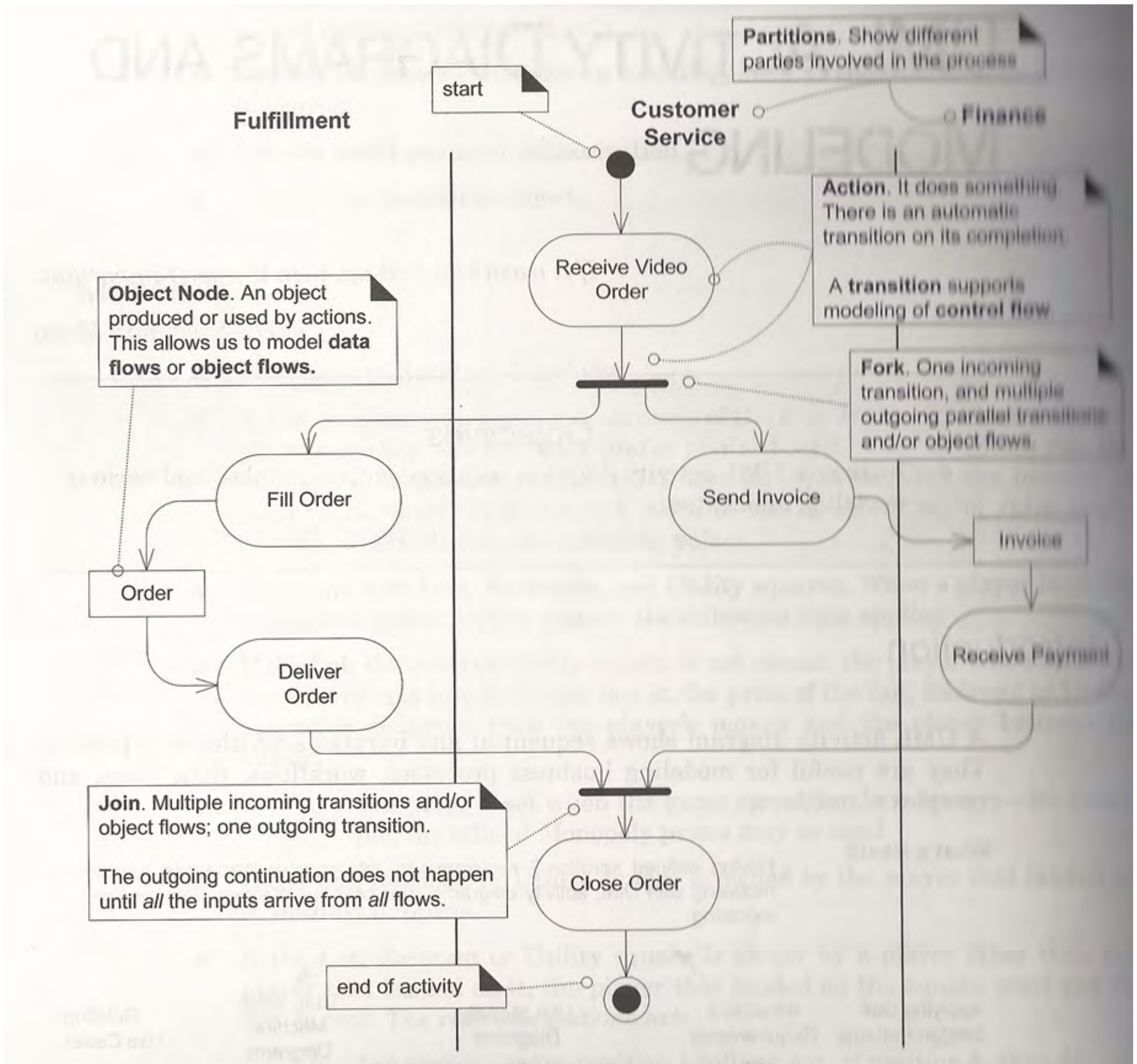

Partitions. Show different parties involved in the process
start
Fulfillment
Customer Service
Finance
Action. It does something. There is an automatic transition on its completion.
A **transition** supports modeling of **control flow**.
Receive Video Order
Object Node. An object produced or used by actions. This allows us to model **data flows** or **object flows.**
Fork. One incoming transition, and multiple outgoing parallel transitions and/or object flows.
Fill Order
Send Invoice
Invoice
Order
Receive Payment
Deliver Order
Join. Multiple incoming transitions and/or object flows; one outgoing transition.
The outgoing continuation does not happen until *all* the inputs arrive from *all* flows.
Close Order
end of activity

Figure 28.1 Basic UML activity diagram notation.

Figure II.2: Activity Diagram Example [23]

The main UML artifact that reflects the overall code structure and relationships between objects and components is the class diagram; specifically, the Design Class Diagram (DCD). It is part of the domain model and provides a graphical representation of each class, their objects or components, interfaces, methods, attributes, and connections [23]. In fact, the class diagram provides so much information about a particular system that it's possible to reverse engineer actual code using only this artifact using software tools such as EA. Figure II.3 shows an example of a shopping software system represented using a class diagram.

2.1.3 Components and CBSE. Component Based Software Engineering (addabbrevComponent Based Software EngineeringCBSE) is very promising for this research and will be referenced several times throughout this paper. Therefore, it is important to provide a general overview of what CBSE is and isn't. It should be noted, however, that this is for informational purposes only; the subject of components in terms of this effort will follow a more informal definition or viewpoint. CBSE and Object Oriented Design (addabbrevObject Oriented DesignOOD) are often confused since they are related and share many core concepts. However, there is a difference and a brief explanation is warranted here.

The roots of CBSE can be traced back to the 1990s. The discipline arose from a research project conducted by Anderson Consulting in an attempt to provide software PnP capability through technological development [4]. Even before this, the foundation for CBSE was laid in response to growing software complexity and size considerations through what is known as middleware. Two of the most important and well-known standards arising during that period are OMG's Common Object Request Broker Architecture (CORBA) and Microsoft's Object Linking and Embedding (OLE). One very important distinction of note is that, although CORBA and OLE can be credited for

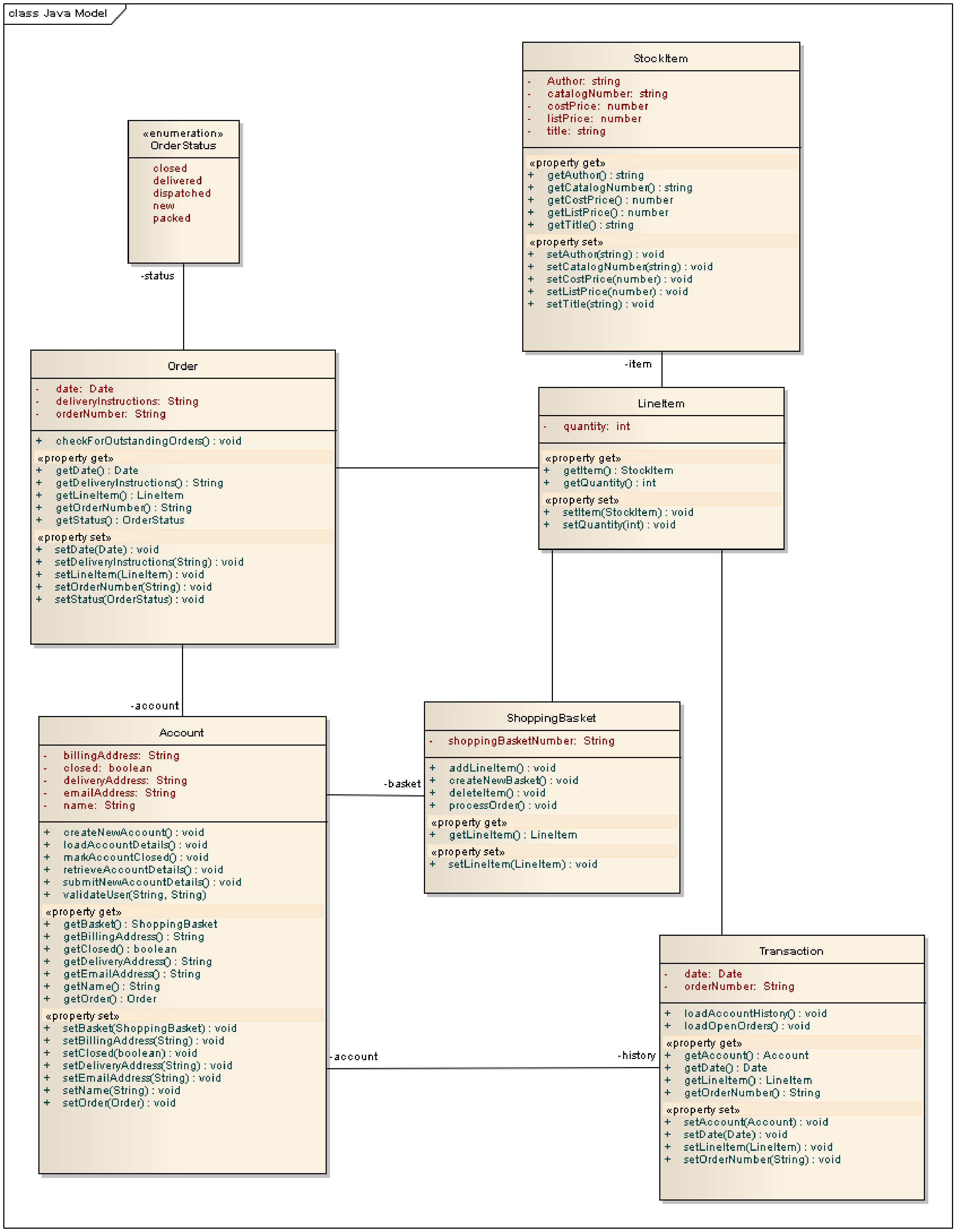

class Java Model
«enumeration» OrderStatus
closed
delivered
dispatched
new
packed
-status
StockItem
- Author: string
- catalogNumber: string
- costPrice: number
- listPrice: number
- title: string
«property get»
+ getAuthor() : string
+ getCatalogNumber() : string
+ getCostPrice() : number
+ getListPrice() : number
+ getTitle() : string
«property set»
+ setAuthor(string) : void
+ setCatalogNumber(string) : void
+ setCostPrice(number) : void
+ setListPrice(number) : void
+ setTitle(string) : void
-item
Order
- date: Date
- deliveryInstructions: String
- orderNumber: String
+ checkForOutstandingOrders() : void
«property get»
+ getDate() : Date
+ getDeliveryInstructions() : String
+ getLineItem() : LineItem
+ getOrderNumber() : String
+ getStatus() : OrderStatus
«property set»
+ setDate(Date) : void
+ setDeliveryInstructions(String) : void
+ setLineItem(LineItem) : void
+ setOrderNumber(String) : void
+ setStatus(OrderStatus) : void
LineItem
- quantity: int
«property get»
+ getItem() : StockItem
+ getQuantity() : int
«property set»
+ setItem(StockItem) : void
+ setQuantity(int) : void
-account
Account
- billingAddress: String
- closed: boolean
- deliveryAddress: String
- emailAddress: String
- name: String
+ createNewAccount() : void
+ loadAccountDetails() : void
+ markAccountClosed() : void
+ retrieveAccountDetails() : void
+ submitNewAccountDetails() : void
+ validateUser(String, String)
«property get»
+ getBasket() : ShoppingBasket
+ getBillingAddress() : String
+ getClosed() : boolean
+ getDeliveryAddress() : String
+ getEmailAddress() : String
+ getName() : String
+ getOrder() : Order
«property set»
+ setBasket(ShoppingBasket) : void
+ setBillingAddress(String) : void
+ setClosed(boolean) : void
+ setDeliveryAddress(String) : void
+ setEmailAddress(String) : void
+ setName(String) : void
+ setOrder(Order) : void
-basket
ShoppingBasket
- shoppingBasketNumber: String
+ addLineItem() : void
+ createNewBasket() : void
+ deleteItem() : void
+ processOrder() : void
«property get»
+ getLineItem() : LineItem
«property set»
+ setLineItem(LineItem) : void
-account
-history
Transaction
- date: Date
- orderNumber: String
+ loadAccountHistory() : void
+ loadOpenOrders() : void
«property get»
+ getAccount() : Account
+ getDate() : Date
+ getLineItem() : LineItem
+ getOrderNumber() : String
«property set»
+ setAccount(Account) : void
+ setDate(Date) : void
+ setLineItem(LineItem) : void
+ setOrderNumber(String) : void

Figure II.3: Example Class Diagram: Shopping System

providing the foundation for CBSE, these architectures are based on object models vs. component models [4].

The distinguishing feature inherent of CBSE versus traditional software engineering practices, including OOP, is that it treats a system as a collection of COTS "components" which are integrated within a particular architecture [18]. The advantages of component-driven design include flexibility, reuse, maintainability, productivity, and distribution. Technically, components can be defined as "units of independent production, acquisition, and deployment that interact to form a functioning system" [13]. It should be noted that these "components," representing significant functional units of a system, are runtime entities; their contexts are limited to when the system is actually running [4][13]. In that vein, the system itself can be viewed as one rather large component [13].

Two graphics follow that serve as an example of UML component diagrams. Both diagrams include the components and interfaces, however, the connections are represented differently within each. The first shows connections between components that are exclusive to this type of diagram, representing a provides/receives relationship. These are sometimes referred to informally as "lollipop" connectors. The second, provided by EA, shows connections that are more in line with what one would observe in other UML diagrams. Both are correct and fully describe the system; the choice is really up to the developer depending on personal preference or even considerations of which might be best suited for the target audience.

2.2 Design Patterns

Keeping with the mindset of building blocks and conceptualizing chunks of code as logical components, Gamma, Helm, Johnson, and Vlissides, commonly referred to as the "Gang of Four" (GoF) borrowed concepts regarding construction patterns introduced by Christopher Alexander in his 1977 book *A Pattern Language:*

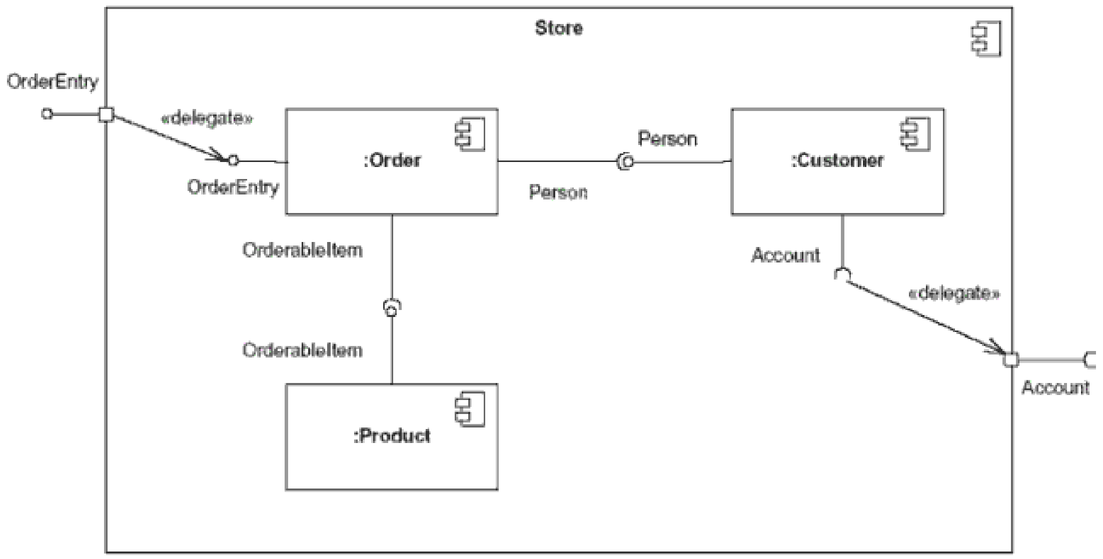


Figure II.4: Example Component Diagram: Shopping System [3]

Towns/Buildings/Construction and generated a new way of writing reusable software in their epic 1995 computing textbook *Design Patterns: Elements of Reusable Object-Oriented Software*. Design patterns have proved extremely beneficial in software design; its positive contributions have perhaps been most noticeable in software maintenance, or life-cycle cost. The GoF book is a good reference to established patterns. These patterns will be used extensively in the research effort and represent the core of how the proposed PnP system will be developed architecturally.

Examples of these patterns are not provided since they will be seen later in the text. However, each pattern is represented in a certain format which is helpful to know upfront in order to aid the reader in understanding them. Every pattern follows a particular template with the following fields: 1) Pattern Name and Classification, 2) Intent, 3) Also Known As, 4) Motivation, 5) Applicability, 6) Structure, 7) Participants, 8)

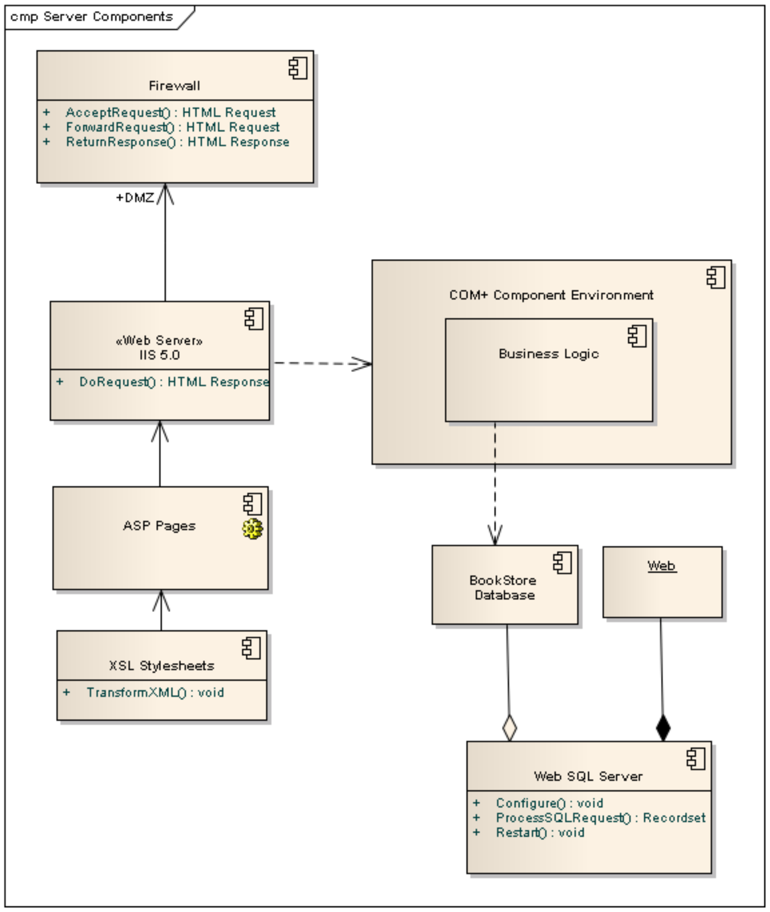

cmp Server Components
Firewall
+ AcceptRequest() : HTML Request
+ ForwardRequest() : HTML Request
+ ReturnResponse() : HTML Response
+DMZ
«Web Server»
IIS 5.0
+ DoRequest() : HTML Response
COM+ Component Environment
Business Logic
ASP Pages
BookStore Database
Web
XSL Stylesheets
+ TransformXML() : void
Web SQL Server
+ Configure() : void
+ ProcessSQLRequest() : Recordset
+ Restart() : void

Figure II.5: Example Component Diagram: Server Components

Collaborations, 9) Consequences, 10) Implementation, 11) Sample Code, 12) Known Uses, and 13) Related Patterns. Table II.1 on page 17 summarizes these fields and briefly describes their purposes [11].

2.3 Player

Player is a software environment designed to provide a language independent communications network for robotics applications [33]. It is in many ways a precursor to ROS. As an open-source project, ROS was heavily dependent upon code reuse to produce proven results in a short amount of time. Amongst others, the code from the Player project

Table II.1: Design Pattern Template Fields

Pattern Name & Classification	Pattern name conveys the essence of the pattern succinctly. Pattern classification reflects scheme.
Intent	Describes what the pattern is supposed to do, i.e., its rationale, intent, the problem it addresses, etc.
Also Known As	Other commonly used names for the pattern if applicable.
Motivation	Illustrates the design problem and how the class and object structures in the pattern solves that problem. This aids in understanding the more abstract description of the pattern that follows.
Applicability	Demonstrates the situations in which the pattern can be applied, how it can address poor designs, and how to recognize those situations.
Structure	Graphical representation of the classes in the pattern using an industry standard notation to illustrate sequences of requests and collaborations between objects.
Participants	Classes and/or objects taking part in the pattern and their responsibilities.
Collaborations	How the participants collaborate to carry out their responsibilities
Consequences	How the pattern supports objectives; identifies tradeoffs, benefits, and dependencies.
Implementation	Indicates pitfalls and other considerations that should be made when implementing the pattern, such as language-specific issues.
Sample Code	Code fragments that illustrate how one could implement the pattern.
Known Uses	Examples of the pattern found in real systems.
Related Patterns	Patterns closely related to the pattern of interest, along with key differences. Lists patterns that should be used along with the pattern of interest, if any.

went directly toward the design and implementation of ROS [36]. Player has been around much longer than ROS, by around ten years in fact; it is still used in many robotics programs worldwide. Player follows a client / server model, which is the foundation of the networking platform it provides [33]. As will be noticeable in later chapters / sections, ROS is modeled more according to a publish / subscribe message passing paradigm.

2.4 Current Sensor Setup

Mr. Jared Kresge, along with others in the ANT center have established a means to run simulations and data collections using multiple sensors for navigation and positioning. The current effort is referred to as Next-Gen VAN for Next Generation Vision Aided Navigation. The name can be misleading, however, as this is meant to serve as an all-source navigation solution. Unfortunately, these methods currently employed take considerable time to set up and are limited in scope; particularly, real-time positioning is desired and not yet possible. Still, the work of Mr. Kresge provides a substantial foundation for this work. Most of the code is OO and has been written in C++ according to the GoF design patterns. This makes altering and/or replacing any portion of the code much easier as it was written with maintenance and refactoring in mind.

The UML for all code parts are not included in this paper; they are much too complex and large to meaningfully fit into anything that provides any sort of insight whatsoever for the reader. However, there are aspects of this work that are worth noting and are thus included here. For example, each time a data collect is to be run, an XML file is created that specifies the sensor configurations that will be used. Configuration files are then used in Player to control and observe the sensors being used. One particular aspect that comes to light here is the flexibility in ROS. Even for code or software that cannot, or is deemed impractical, to refactor into ROS launch files, Player can be used within ROS using specific Player stacks that are available. Simply put, the software that is written to be run using Player can be run in ROS by utilizing the Player packages already available.

To demonstrate the patterns that have already been discussed being used for this effort, an example graphic of the UML serves as a very good example. As can be seen in Figure II.6, which shows the objects written as patterns, the overall system is arranged quite logically in a traditional OOD hierarchy. As can be seen from the figure, much work has already been completed in terms of the software for gathering positioning data from various sensor testing configurations. However, it is repeated that these data collections have been primarily limited to post-processing of data only. Some real-time handling of the data is available with Next-Gen VAN, however, any changes to the sensors require extensive setup and configuration time, which is certainly not ideal. A real-time, adaptable sensor integration platform is desired to allow changes in hardware to be made on the fly. This type of platform would be much better suited to address environment changes, allowing data to be selected from various sensors according to the needs of an application rather than being fixed upfront.

2.5 IEEE 1451.4

The idea of self-describing behavior existing within sensors used for N&P solutions is key in this research, and in PnP in general. This is required for basically any type automated hardware recognition because a system must have a way to identify what is "plugged in" and other information allowing for it to be set up and configured for use. Using wireless networks as an example application, Michael Dunbar explains "The network host must know which sensors are connected, which sensors are within range...and when new sensors are added to the network." He then lists the following two characteristics a PnP sensor network will have: 1) a standardized architecture and 2) a self-identification protocol [8]. IEEE 1451.4 defines a mechanism for adding self-describing behavior to transducers (certain types of sensors) with an analog signal

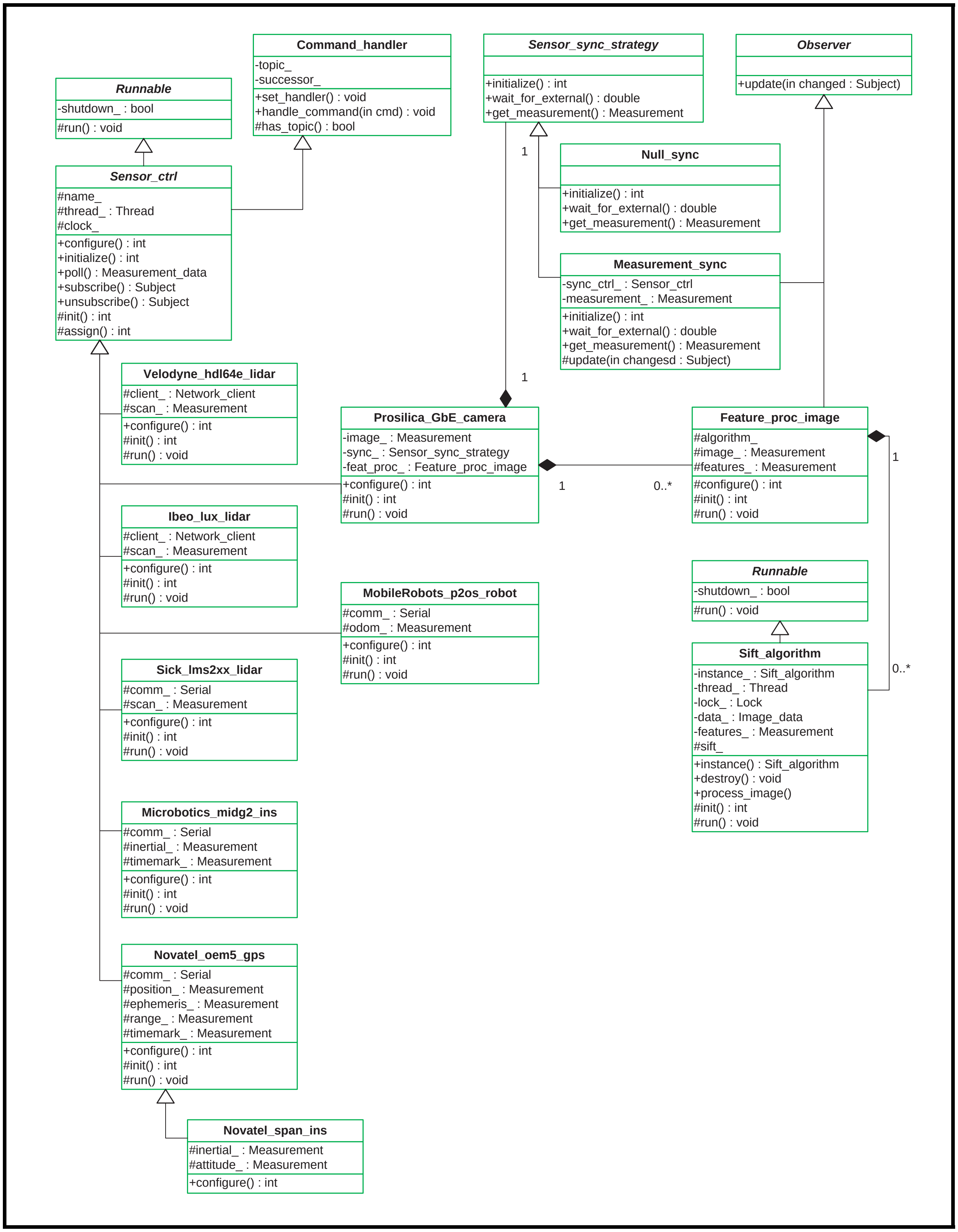


Figure II.6: Portion of Next-Gen VAN Class Diagram

interface. This provides the common, agreed-upon way to integrate sensors within a network desirable for PnP.

This standard is implemented through requiring each compliant sensor to contain self-identifying information in a standard format. Specifically, these sensors utilize what are known as Transducer Electronic Data Sheets (TEDS). TEDS is the principal characteristic of an IEEE 1451.4 PnP sensor [34]. TEDS includes all of the information needed to describe and use the sensor before making measurements; at a minimum, this includes the manufacturer, model number, and serial number for the transducer [34]. An example graphic of a TEDS sensor is shown in Figure II.7 on page 21. Also, an example of what is contained within TEDS and its structure are found in Tables II.2 and II.3.

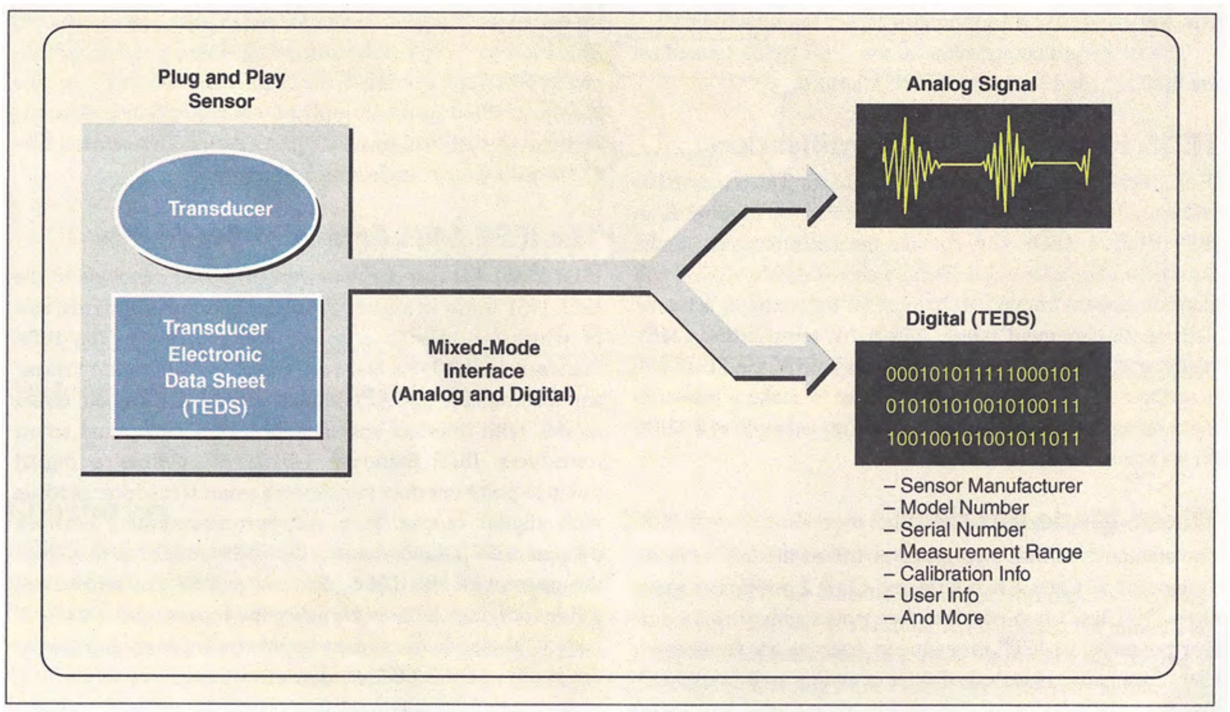
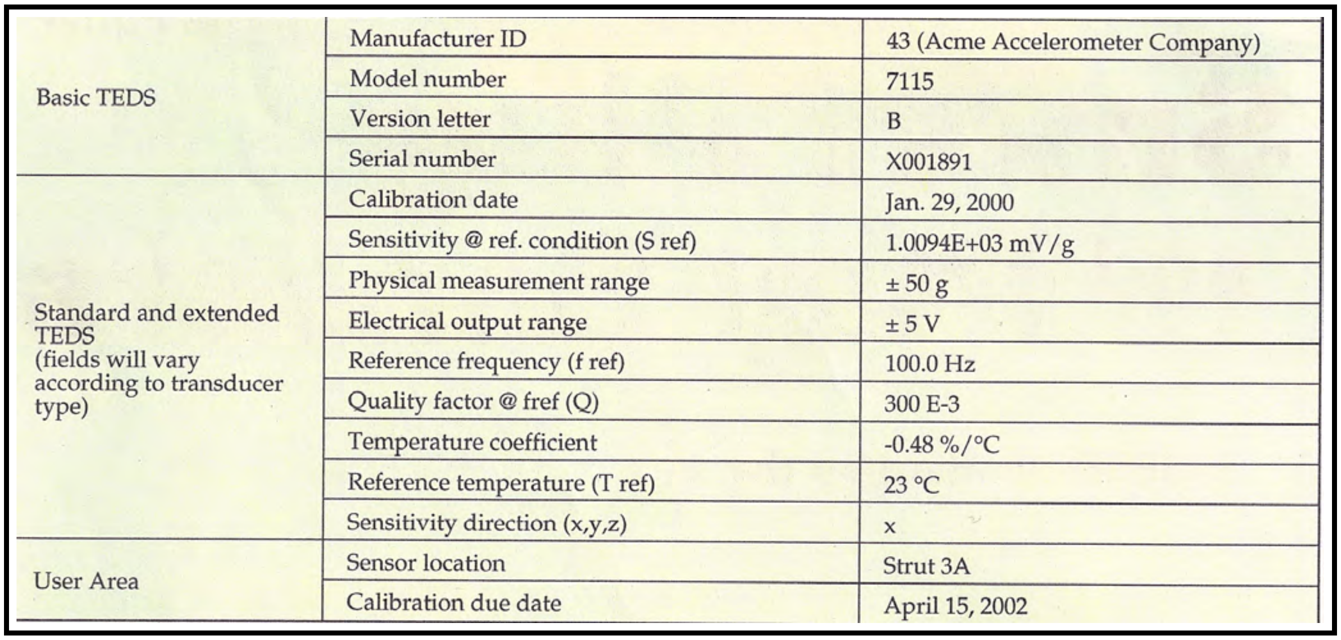


Figure II.7: PnP sensor with embedded TEDS information [34]

Table II.2: Example TEDS for an Accelerometer [34]



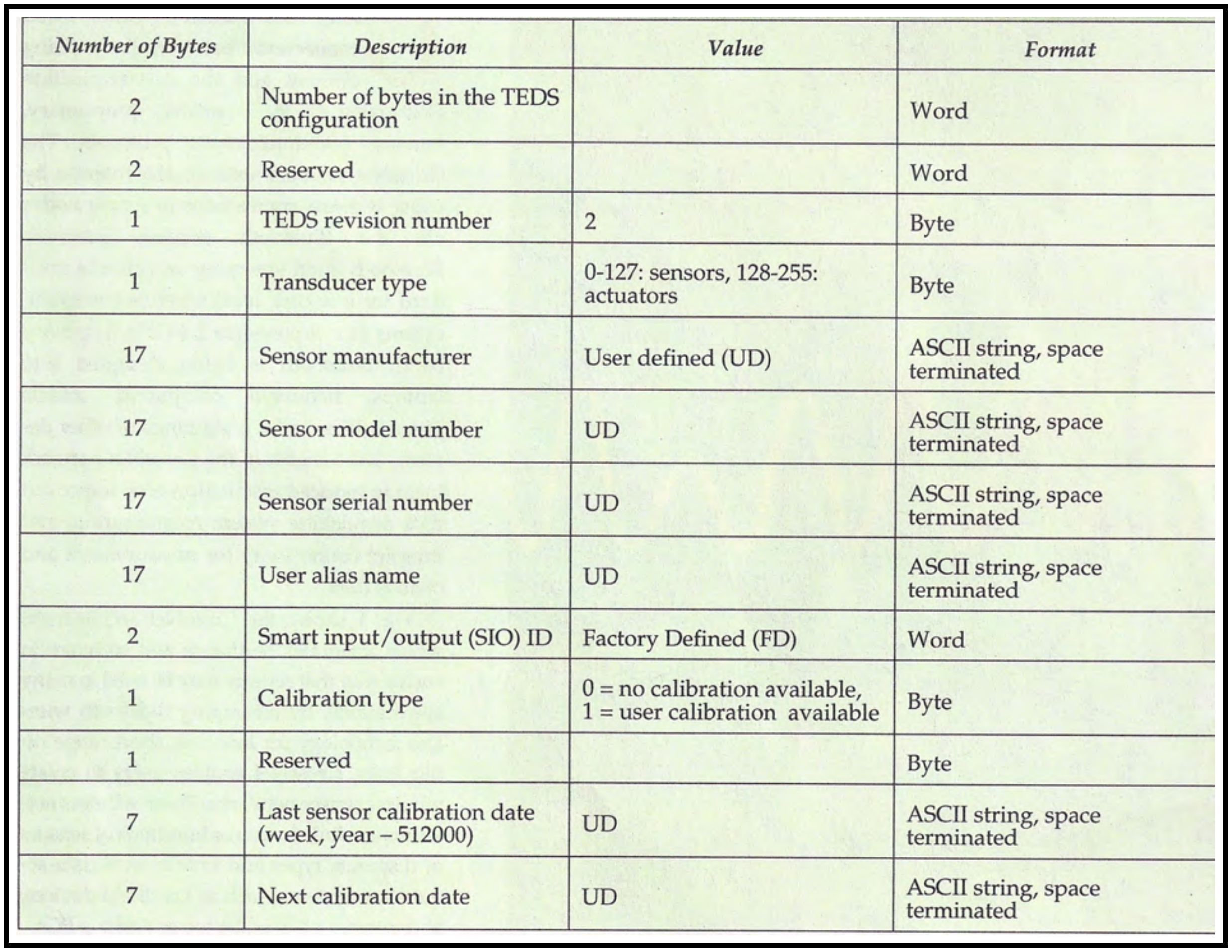
Basic TEDS	Manufacturer ID	43 (Acme Accelerometer Company)
	Model number	7115
	Version letter	B
	Serial number	X001891
Standard and extended TEDS (fields will vary according to transducer type)	Calibration date	Jan. 29, 2000
	Sensitivity @ ref. condition (S ref)	1.0094E+03 mV/g
	Physical measurement range	± 50 g
	Electrical output range	± 5 V
	Reference frequency (f ref)	100.0 Hz
	Quality factor @ fref (Q)	300 E-3
	Temperature coefficient	-0.48 %/°C
	Reference temperature (T ref)	23 °C
	Sensitivity direction (x,y,z)	x
User Area	Sensor location	Strut 3A
	Calibration due date	April 15, 2002

TEDS does not satisfy the need for self-describing sensors for this research since the number and type of IEEE 1451 compliant sensors is quite limited. In fact, none of the sensors that are likely to be used with an N&P system are IEEE 1451 compliant. What the standard does do, however, is demonstrate that this information can be included natively within a sensor and lays a foundation for future work. It is unlikely during the course of this research a new standard will be developed enforcing this behavior for all sensors, including those used for guidance, positioning and navigation. Looking to the future, however, with the success of endeavors such as this one, it is almost assuredly inevitable.

2.6 Toward Plug-n-Play

There are three essential elements to an effective PnP sensor system: 1) sensors demonstrating self-describing behavior, 2) real-time tracking and status of health (SOH) of all sensors, and 3) platform independence in terms of network communication. The third element is already somewhat addressed by the software currently employed within

Table II.3: Example of Fields within a TEDS [8]



Number of Bytes	*Description*	*Value*	*Format*
2	Number of bytes in the TEDS configuration		Word
2	Reserved		Word
1	TEDS revision number	2	Byte
1	Transducer type	0-127: sensors, 128-255: actuators	Byte
17	Sensor manufacturer	User defined (UD)	ASCII string, space terminated
17	Sensor model number	UD	ASCII string, space terminated
17	Sensor serial number	UD	ASCII string, space terminated
17	User alias name	UD	ASCII string, space terminated
2	Smart input/output (SIO) ID	Factory Defined (FD)	Word
1	Calibration type	0 = no calibration available, 1 = user calibration available	Byte
1	Reserved		Byte
7	Last sensor calibration date (week, year – 512000)	UD	ASCII string, space terminated
7	Next calibration date	UD	ASCII string, space terminated

the ANT center (see Section 2.4), which follows a publish-subscribe message passing scheme within a client-server paradigm. If these terms are not familiar to the reader, they will be introduced in subsequent subsections. This type of software abstraction provides natural clues into how PnP may be implemented in various systems for multiple applications. In fact, there have been considerable studies conducted which directly lend to this rationale; some of which will be introduced in subsequent sections.

2.6.1 Client/Server Abstraction. Client/Server is an abstraction involving two primary actors: the client and the server. Typically, an entity, such as an object can act strictly as a client or server, or may act as both. Further, an entity may be serving as a client at certain times and as a server at others. SearchNetworking.com defines Client/Server as “the relationship between two computer programs in which one program, the client, makes a service request from another program, the server, which fulfills the request.” The site further explains that even though this can be used on a single location, such as one computer, it has far greater implications in networks. In a network, this message-passing relationship affords a natural way for different programs or entities to pass data from one location to another [41].

2.6.2 Web Services. Web service is a common means of implementing the Client/Server model. It is natural to think of a web service in terms of the worldwide web; undoubtedly that is likely how it got its name since many internet applications utilize web services. World Wide Web Consortium (W3C) defines the Web Services architecture as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format...” [47]. Therefore, the use and application of a web service is not limited to the worldwide web; it can be used in any network. One way to look at a web service follows directly in line with the Client/Server model; that is, as a distributed application [6]. This leads to the discussion of Service Oriented Architectures (SOAs) which represents the primary platform in which web services will be employed to address the PnP problem.

Web services utilize message passing schemes that are language independent. This introduces the concept known as middleware. For web services, the middleware is fully described with XML, Simple Object Access Protocol (SOAP) or Web Services Description Language (WSDL). WSDL “tells a service requester where a Web service is located and how to use it” [6]. IBM defines Service Oriented Architecture as “an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications” [19]. Web services lend themselves directly to SOAs.

2.6.3 Client/Server vs. Publish/Subscribe. Message-oriented middleware, as introduced above, has spawned the publish/subscribe messaging scheme. Conceptually, the publisher can be viewed as the sender and the subscriber as the receiver. The corollary here with Client/Server is obvious — the client is the subscriber and the server is the publisher. There are subtle differences in client-server and publisher-subscribe but the two are virtually interchangeable in the context of this paper. Typically, publisher/subscribe is considered more loosely coupled than a general client/server architecture.

2.7 Reflection Architecture

Ippolito *et al.* present a component-based PnP solution to prototyping and testing unmanned vehicles in their Reflection Architecture (RA). RA is “a real-time component-based plug-and-play architecture for the development of embedded vehicle systems.” This was introduced to allow rapid prototyping and development in the field of embedded technology. A major aspect of RA, as should be the case with a PnP system in general, is that it provides no implementation details regarding the hardware it is addressing. The components themselves are responsible for describing their objects and each contains its own implementation. Because of this, the system itself doesn’t need to know anything about how the sensors are described or how they work. All that is required is that data is passed according to the interface, enforcing the concepts of language and domain independence. This presents another major benefit to RA, which is the ability to simulate portions of hardware without affecting the performance of the overall system [14].

RA is currently being used in real-world robotics applications, such as the CMU West MAX Rover and NASA Exploration Aerial Vehicle Platform, where it serves as the core embedded architecture. This makes RA unique in that is truly a PnP system for robotics sensor integration that is already specified, implemented and tested in major research programs. This cannot be said for other background work that has been, or will be, introduced in this chapter. The GOF design patterns introduced previously are adhered to for the software implementation, with method metadata structures closely mirroring the command pattern and a facade interface for grouping similar attributes as one logical entity [14].

2.8 Sensor Abstraction

A major part of almost any proposed PnP system is the ability to abstract certain aspects of the software being developed. The utility of such a technique is perhaps even more apparent in a real-time PnP system for navigation and positioning since it is, by its very nature, a context-aware mobile software system. This brings about many unique challenges, some of which will be highlighted in this section. Much of the focus will be centered on integrating components, or in this case sensors, since a system is comprised of a variety of them and the system should be as scalable and nonspefic as possible.

For a vehicle comprising a robotics platform, which is the case for this and many navigation and positioning applications, there are often several people from varied backgrounds and disciplines working toward a common goal. Each will work on his specific area of the problem, coming up with a solution that works, and much time and effort is spent later integrating it with the works produced by others. For those familiar with the challenges this can present, an agreed upon interface is often specified upfront as to alleviate complications further into development. However, the extent of scalability and robustness is often sacrificed in such cases in favor of simplicity. This can be captured in a simple abstraction; an example of one possibility is shown in Figure II.8.

In order to assure maximal cohesion and loose coupling, steps must be taken to allow changes and enhancements to specific components that will transcend the entire system. In other words, changes shouldn't be done in isolation and a perspective on the entire system should be maintained at all times. This is where abstraction can be beneficial,

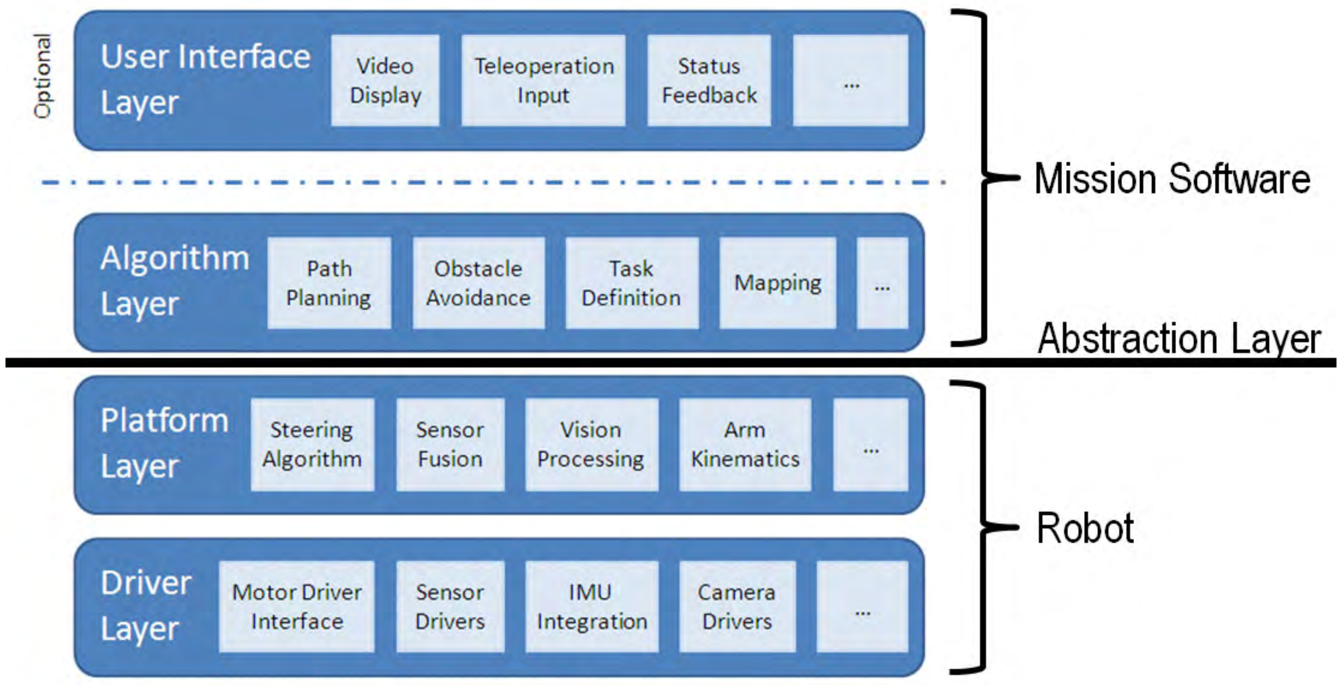


Figure II.8: Example of System Integration Abstraction [35]

providing a software layer that allows for dynamic implementation solutions. According to Powell and Muecke, who are quite experienced in the test and measurement industry, "A requirement for this is to use system-level software tools and software architectures that lend themselves to cross-disciplinary development" [35]. ROS is one such tool which is why it was chosen for this particular research research study. In fact, the big takeaway with using an abstraction interface is two-fold: 1) higher-level software is unaffected by changes in lower-level sensors and 2) sensor drivers remain unchanged with changes in higher-level software. One other major implication of an abstraction layer is the ease in which one could simulate hardware that isn't present by programming according to the sensor Application Programming Interface(API) [35].

2.9 Jini

Jini is a systems architecture developed by Sun microsystems and introduced in 1999. It was designed to federate users and resources with the goal of creating a more flexible and dynamic network based on components and services [42]. Not surprisingly, Jini is very much intertwined with Java as both are Sun products. Java and Jini, together with Sun's Remote Method Invocation (RMI) technology provides the basis of a service architecture adaptation for components and services connected over a network. Like CORBA and OLE, it is also middleware that can be used to suit the needs of a given distributed systems project. In fact, Jini functions as a primary architecture within a service-oriented spacecraft architectural model proposed by AFRL: "Java-based Plug-N-Play (Flight) Control Systems for Responsive Space." [30]

Jini is referenced in this section as it has been used in a specific Air Force attempt at software PnP for satellite systems, which involves sensors. Although Jini is not used in this particular effort, it demonstrates how middleware is used to resolve network control functionality in such a system. For example, Figure II.9 provides a depiction of how Jini is used as middleware in a client/server connection scheme.

While Jini is not used for this particular research, a well-developed systems architecture such as this might be considered for future work. The scope of this thesis precludes the level of features and complexities that would make inclusion of such an architecture worthwhile.

2.10 Context-Aware Mobile Software

This section introduces the research conducted by Fortier, Rossi, Gordillo, and Challiol entitled "Dealing with variability in context-aware mobile software." It is specifically mentioned here as it will be referenced explicitly in later sections. This

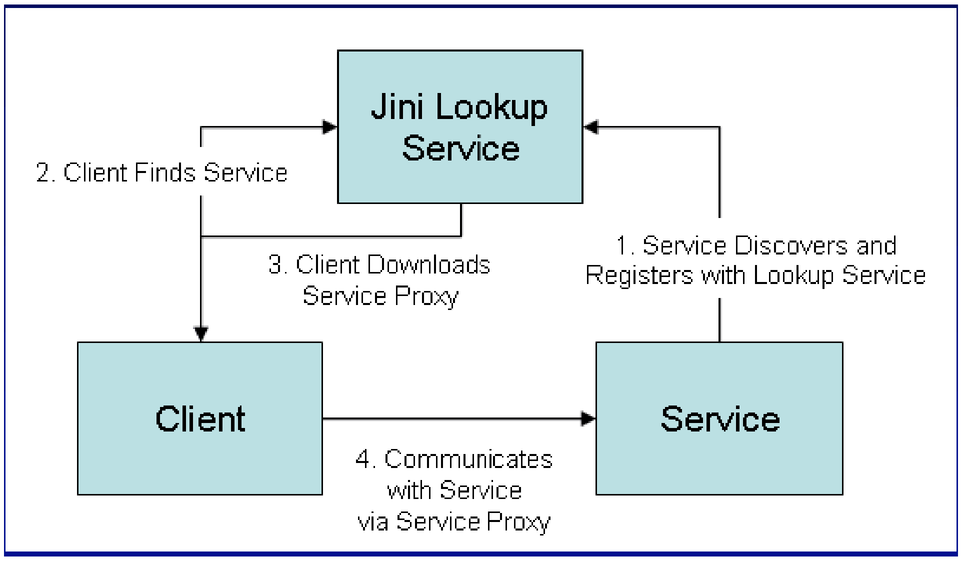

Jini Lookup Service
2. Client Finds Service
3. Client Downloads Service Proxy
1. Service Discovers and Registers with Lookup Service
Client
Service
4. Communicates with Service via Service Proxy

Figure II.9: Jini Discovery, Join, Lookup, Service Invocation sequence [30]

section ties in directly with Section 2.8, "Sensor Abstraction," as it discusses a way to abstract out complexities using carefully developed models.

2.10.1 Motivation. Mobile context-aware software represents a unique and challenging problem in the field of software development. Not only does it often involve large numbers or components with rich functionality, there is the added, and rather daunting, complexity introduced by its intrinsic, dynamic nature. That is, the software itself must be capable of behaving a certain way in response to the context of its user. For example, a piece of software acting as a travel assistant will need to provide Location Based Services (LBSs) which change depending on the user's physical location [10]. In the context of this particular research, a user's positioning algorithm will change based on the physical location of a vehicle, what sensors are available to it at a particular time, how accurate its measurement data is, etc. Fortier *et al.* developed their own methodology for solving these types of problems and laid a foundation upon which others can tailor to their own software development needs. Because of the obvious implications in terms of navigation and positioning, this paper serves as a major reference to part of the work introduced in Chapter 3.

2.10.2 Domains and Variability. The framework involves two major domains in terms of the architecture which the authors refer to as the "application" and "adaptation" domains. As they put it, "...we would like to have an architectural infrastructure that provides the most common abstractions (such as sensors) and behaviours (such as different types of adaptation)...,leaving the application designer with the task of developing 'only' the application's specific functionality." In essence, the application (the what) represents the core functionality or feature whereas the adaptation is how the behavior modifies it to meet a particular context (the how). For example, travel assistance software may use GPS to resolve a user's location which falls under the application domain. With the position obtained by the GPS data, the travel assistance software could then provide LBSs (e.g., displaying the nearest gas station, ATM, etc.) which would fall under the adaptation domain.

Three major areas are characterized as the main variability issues with mobile context-aware applications: 1)Dynamic Context Model, 2) Sensor Support and 3)Domain-specific adaptation. There are also two interdependent sources for variability that must be addressed in distinct stages. Specifically, variability is addressed in different ways during design time and run-time. At design time, all of the possible variations must be considered upfront and therefore is static in the sense that it cannot be changed during run-time. Run-time variation, however, is dynamic in nature since different applications and adaptations are limited only by what has or has not been addressed at design time. Using navigation and positioning as an example, the array of possible sensors that may be available or needed must be specified upfront at design time. During run-time, however, these sensors can be used in whatever combination, configuration, or in as many different ways as is required according to the specific context. For instance, if a vehicle moves from an indoor location to an outdoor one, GPS can be brought online and used providing a GPS sensor was specified at design time. This scenario also serves to highlight the importance of run-time discovery and reconfiguration for a mobile context-aware software system [10].

2.10.3 Modeling. This section introduces the models developed in the Fortier study that are most relevant in this research; that is, those that related to sensors and sensing algorithms. The context model derived by the authors is based on two basic concepts: aware objects and context features. An aware object can be conceptualized as a software entity that checks for changes within the system. For example, an aware object knows when a sensor has been added or removed from the system. For design purposes, the aware object is modeled as an abstract class and instantiated when it is to be used for a particular purpose. This is where the context feature fits in by allowing a concrete realization of a particular context model. It "...holds the current value of a context property and informs any interested part whenever that situation changes." Therefore, for any context model to be realized, there must be an aware object and a context feature for each context aspect. [10]

An example of a simple context model can be found in Figure II.10. This model is for tracking a user's location. The diagram depicts an aware object and one context feature (for location).

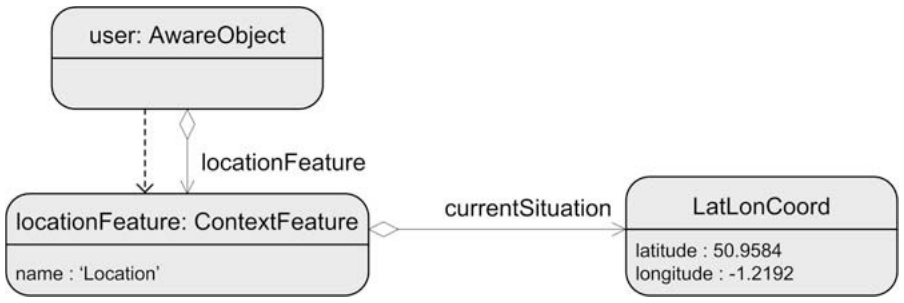


Figure II.10: Example of a simple context model [10]

It is worth revisiting the variability loops and the two stages in which they are addressed: design time and run-time. During design time, the loop must be closed by addressing all of the foreseen variation points, or context features, that will be bound at run-time. In this stage, the aware objects and their contexts are defined by designing and implementing each context feature. On the other hand, when the software is running a context feature may or may not be active. For example, a GPS unit may not be in use due to the fact that the test platform is operating indoors during a particular period of time. As the authors put it, "...we consider each context feature as a variation point of an aware object; each context feature is closed at design time and is treated as an optional variant that is re-evaluated in run-time." In the case of the GPS example, this can be accounted for at run-time by simply removing that particular context feature, since it isn't of use in such an environment. This capability is afforded by the manner in which aware objects provide messages for context management, such as "addFeature" and "removeFeature" messages.

Figure II.11 represents the ContextFeature hierarchy. There are several different types of features in this graphic that have their own definitions, however, explaining them doesn't provide much added benefit for our purposes. The main focus is on how features and aware objects are structured within the model.

2.10.4 Sensor Support. The subject of sensor support is addressed explicitly in the paper and is not repeated in full detail here. If more information is desired, the reader is directed to p. 925 of the referenced journal article. With all of the contexts resolved for the system, it is also necessary to handle how external data will be added and used within those contexts. This is done in an OO fashion, using a specified communication interface according to a publish-subscribe message passing scheme. An example instance diagram for a sensing layer is shown in the following figure.

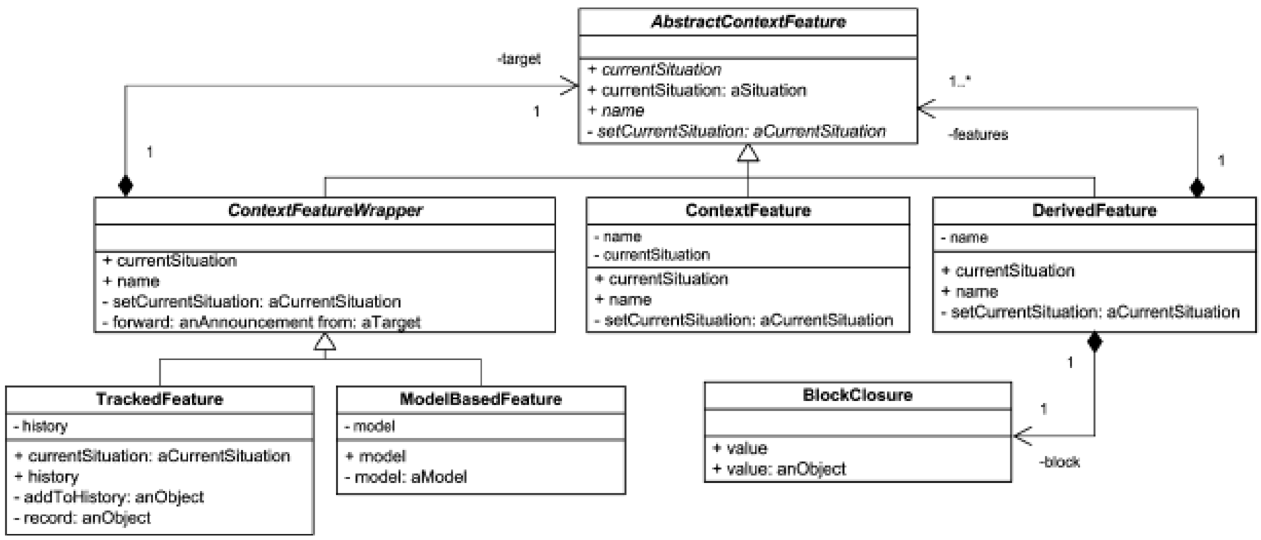


Figure II.11: ContextFeature hierarchy [10]

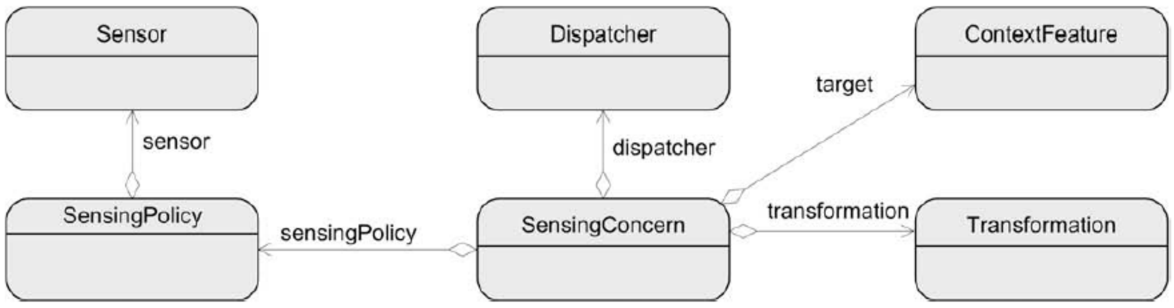


Figure II.12: Simplified instance diagram of the sensing layer [10]

Of course this representation is just one of many possibilities in addressing the inclusion of a broad range of sensors. The one shown here addresses error-checking early on through the Dispatcher. After the SensingConcern receives the sensed values, the Dispatcher will determine whether or not it is valid. It is then passed on through the rest of the system according to the context it is addressing, transforming the data into a proper format if required.

To solidify the explanation through an example, two more graphics are included. The first, Figure II.13 on page 35 is a class diagram of a sensing architecture. This depiction can be followed fairly easily through simple observation. It captures the aware objects, context features and sensing policies previously discussed, how they fit together structurally, and how variation is handled at design time. The next, Figure II.14 on page 36, is a more specific example showing how a GPS sensor can be used in this model to receive sensor data. It captures context specific handling of the data and demonstrates communication between sensing policy, dispatcher, and the sensor itself.

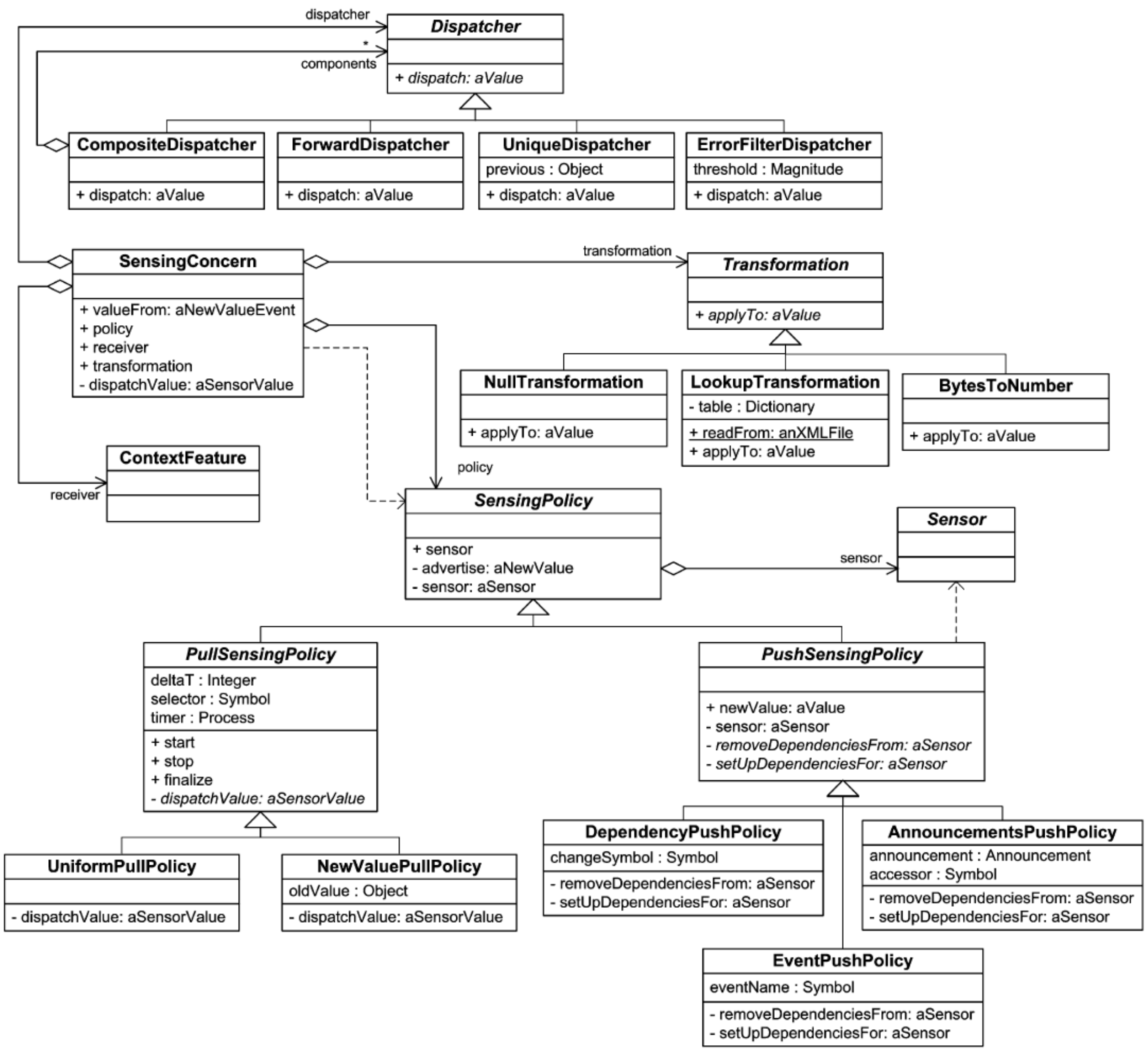


Figure II.13: Example sensor class diagram [10]

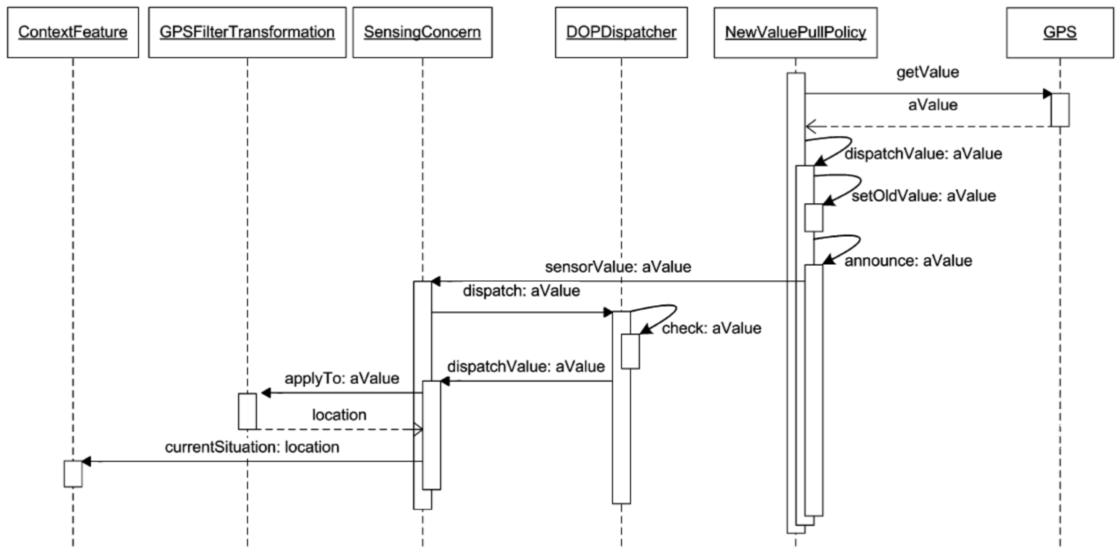


Figure II.14: Dispatching a GPS signal [10]

III Methodology

This section presents a real-time PnP solution for handling multiple sensors used in navigation and positioning applications. This differs from what has been developed within the ANT center with the Next-Gen VAN project in that this particular work focuses on autonomous configuration of sensors and the capability to use them in real time. As has been mentioned in previous sections, current efforts have been quite limited in terms of hardware and software scalability. In particular, much of the work depends on post-processing data that has been collected during a particular test run. At the same time, algorithms that are implemented for real-time data use are preconfigured with a particular sensor configuration that has been bound in the software upfront and cannot be changed afterward.

The proposed solution is still design-time static in that sensor variability should be addressed *a priori*. What's different is that all of those sensors can be configured, begin collecting data and publish that data to an aware object which, in turn, handles those data requests. Any sensors that are added, removed, or become otherwise unusable are tracked by this object, thus providing an always up-to-date repository of data upon which an application can request any combination of streams. These concepts are explored more thoroughly throughout the rest of this paper.

Another way in which this particular architecture is different from that which has already been developed is that it is more generally focused and can be extended to other applications. For instance, the Next-Gen VAN software includes not only a platform for gathering data but also the Kalman filtering required for adhering to a particular positioning algorithm, whatever that might be. In this approach, the handling of sensors and their use in resolving positioning are treated as two distinct chunks of the overall software architecture. This is a logical way to treat such a system since the software responsible for configuring a sensor and receiving its data shouldn't require any sort of knowledge about how that data will be used afterward.

3.1 Overview

To start, it is of vital importance to state upfront the limitations and scope of this PnP solution. In the proposed solution, the sensor is brought on line and begins to take measurements. The data is then output or stored and can be used in whatever way a given application requires. Therefore, the portion of the software responsible for using the data, which represents the user layer, is included in the architecture only to demonstrate how a complete PnP system for navigation and positioning could be realized. All of the actual software developed will be for the application/system layer. That is, all of the coding is geared toward configuring a sensor when it is introduced to the system and handling its data. The "computer software" and "operating system" entries in Wikipedia provides a very simple graphic of software layers which is included here in Figure III.1.

Of course the eventual goal is for universal PnP: that is, for any sensor to be added or removed from a system at any given time and for the navigation and positioning algorithm to be able to reflect this accordingly in real-time. Unfortunately, the technology isn't quite to the point that allows this to be achieved in full without adding extreme development complexity. One reason for this is in how the variability is handled during development as indicated in Chapter 2, which is in part a by-product of the fact that sensors are passive devices. In turn, self-identification and self-loading drivers that are apparent in devices such as USB peripherals for personal computers cannot be expected natively. The other reasons will be discussed in subsequent sections of the text but primarily, there wasn't enough time and other resources that allowed for this. As such, this solution leans heavily toward that of a proof of concept as far as the actual experimentation is concerned. Three

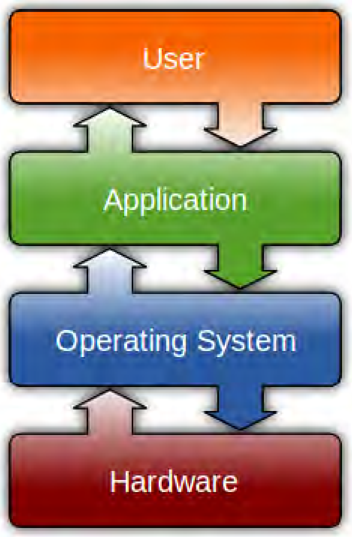


Figure III.1: Graphical representation of software layers [46]

sensors were used: two using a serial-to-USB hardware interface and Ethernet for the other. A listing of the sensors can be found in Table III.1.

Table III.1: Listing of Test Sensors Used

Manufacturer	**Sensor Type**	**Model**	**Hardware Interface**
SICK	LADAR	LMS200-30106	Serial-to-USB
Microstrain	AHRS	3DM-GX3-25	Serial-to-USB
Prosilica	Vision Camera	GE1660C	Ethernet

In order to achieve true PnP, there are three key system level requirements. The first is self-describing behavior, which is required in order to allow the host system to recognize when a new piece of hardware has been connected and what it should do with it.

This implies the sensor must be an active device in terms of software, which may be rather different than what one would expect it to be. For example, a sensor may be actively generating pulses and detecting changes based on differing return signals, however, this is still passive in terms of the host software. This presents a considerable challenge to achieving a PnP sensor environment. Short of IEEE 1451, there really isn't a standard for how a sensor interfaces with any other hardware's software unless it was developed for a specific purpose. None of the sensors used in the ANT center are IEEE 1451 compliant, nor are most sensors used for navigation and positioning or robotics for that matter.

This chapter begins with an overview of the test setup, discussing the hardware platform and various considerations that were made to achieve the desired effect. The discussion then turns toward outlining the core components of a PnP architecture and follows directly into a conceptualized component-based software development approach to this particular research problem. The rest of the chapter addresses specific solution approaches. Along with the description of the setup, background information about certain concepts that have yet to be provided will be summarized. This is done for two reasons: to allow the reader to understand certain key concepts of which he might not be familiar and to explain the rationale behind the chosen approach.

3.2 Hardware Platform

The main hardware platform that was used for this research is the MobileRobots PowerBot, although some of the tests relied upon Universal Robot Description Format (URDF) models to quickly test some of the ROS specific launch files (predominantly for the p2os stack). Initially, the work was attempted on the Pioneer P2AT, which would have been easier to work on based on its pervasiveness is in the robotics community, thereby affording a higher knowledge pool to draw from in general than for the PowerBot. Unfortunately, the legacy hardware present on the robot itself was not current enough to run the up-to-date ROS builds used in the experimentation.

The SICK laser used for the experimentation was already present on the robot and is interfaced using a custom configuration board provided by the manufacturer. The other two sensors were connected to the side input ports of the robot. Only one of the USB ports was needed and is arbitrary as to which is chosen to interface the MicroStrain AHRS. There is only one available Ethernet port so naturally this is the one that was chosen for the Prosilica GE1660C camera. The AHRS requires an external power source to run, which is plugged into a standard 110V AC outlet. Since the application would eventually require mobility, this power option would need to be dealt with sooner or later. One option is to procure the USB version of the hardware or to obtain a USB power adapter rated for this device. The same is true for the Prosilica camera regarding an external power source requirement. This has been resolved in the past through the use of mobile power adapters attached to the robot itself in the case of the Pioneer P2AT.

The PowerBot configured with all three sensors is shown in Figure III.2 on page 42. The SICK camera can be seen in the graphic mounted to the front center of the robot. The Prosilica is directly on top of the robot in the center above where the SICK is located. Finally, the MicroStrain and SerialGear serial-to-USB converter are located on the top of the robot on the right-hand side. Off to the right and down from the AHRS is where all of the connection ports are located (above where the open compartment door is visible).

Most of the hardware within the PowerBot is not factory installed equipment. The hardware was upgraded to provide more computational power and modern day interfaces, such as USB 3.0. The host OS is Ubuntu 10.04, Lucid Lynx, which is the current Long-Term Support (LTS) version of Ubuntu. Ubuntu is currently the only fully-supported operating system for use with ROS. Also, all of the recent versions of ROS were developed for Ubuntu 10.04 and future releases (currently through 11.10).

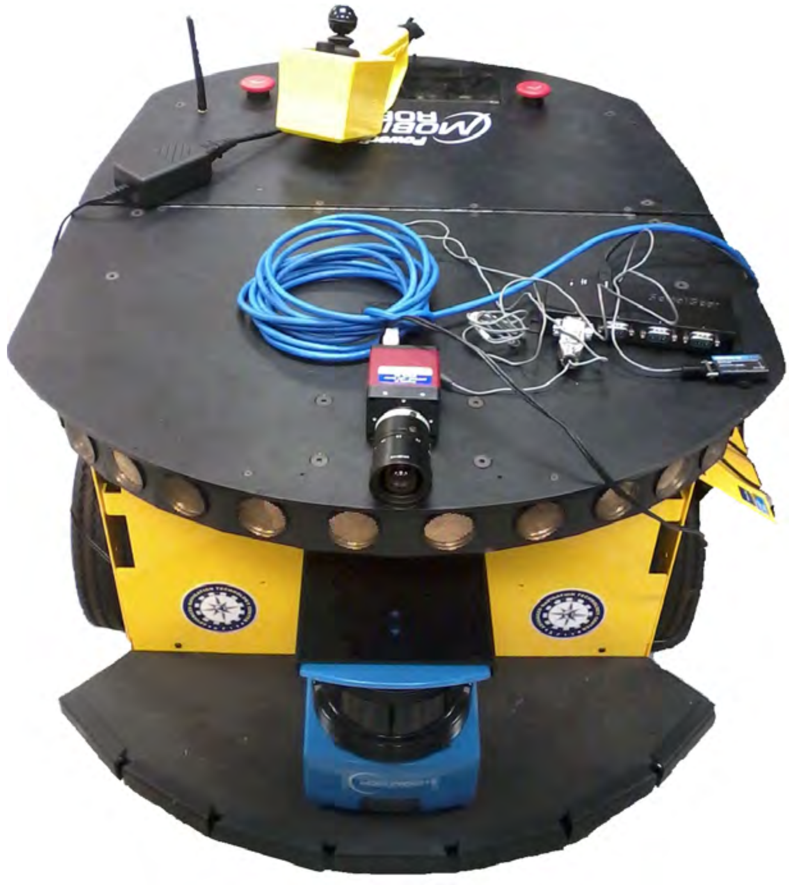


Figure III.2: MobileRobots PowerBot Fully Configured w/ Test Sensors

3.3 ROS Contributions

One of the most significant advantages to using ROS is a direct result of the direction in which the robotics community has shifted. That is, people are aware of the numerous sensors and platforms currently in use and have since become much more focused on scalability and reusability of software and hardware for various applications. This is a need upon which ROS has been focused and has since seen widespread use and development throughout the open source robotics community. Universities and individuals have developed ROS drivers, utilities programs, and launch files for virtually every robotics sensor available, to include some which may not seem traditional in that arena such as the Microsoft Kinect gaming sensor. With this pool of COTS software, virtually all of the requirements for configuring and gathering data with the sensors used in this research are satisfied by simply using them with ROS. There is somewhat of a learning curve with ROS, however, and some time and effort is required resolving all of the package and system dependencies for a particular application. This is addressed for the three sensors used in this research in the following subsections.

3.3.1 MicroStrain AHRS. The sensor used in this experiment is newer than that currently listed for support in ROS. However, this issue has been addressed in the ROS forums and has yielded an update to the legacy software allowing it to be used with the 3DM-GX3. Once the drivers are executed, the data is available by subscribing to the corresponding rostopic. The data is output to stdout, or terminal, but can easily be saved to a file or otherwise used directly in some other application as long as the data is consistent with its specified input.

3.3.2 SICK LADAR. The SICK LMS200 is one of the most common navigation sensors currently in use for robotics applications. Not surprisingly, it is fully supported in ROS; there is much information available to anyone wishing to use this particular sensor in whatever application is desired. However, there is a known issue with MobileRobots machines and their compatibility with what is currently available with ROS. This is due to the special converter board that is native to these machines and has seen some resolution with hard coded changes to the sicktoolbox package. After the changes have been made to the source code, the package must be recompiled and visible within the ROS package path.

3.3.3 Prosilica Camera. As stated earlier, the Prosilica 1660GE is used for this experiment. ROS can be used to both find the cameras current IP address by running the ListCameras executable in the Prosilica camera package. With this information, the Ethernet port can be configured for the camera; since this research does not require any other devices with an Ethernet interface, it can be configured as a dedicated Ethernet port. This is not specific to ROS and is performed in Ubuntu the same as it would be for use in some other application. There is already a launch file written that will generate a camera node which needs to be edited to reflect the cameras correct IP address obtained previously. After editing and executing the launch file, the images are output to a file. These can be accessed easily through ROS by running the “image view” executable from the “image pipeline” package. The “image view” provides the functionality that can be found with Allied Vision’s “SampleViewer, which is the manufacturer-provided image viewer.

3.4 PnP Requirements

In order to achieve true PnP, there are three key system level requirements. The first is self-describing behavior, which is required in order to allow the host system to recognize when a new piece of hardware has been connected and what it should do with it. The second, which goes hand-in-hand with the first, is a means for discovering where and how the sensor is connected to the system. Finally, once the sensor has been successfully connected to the system, it must be configured for use by the host OS by loading the proper drivers. Each of these are explained in more detail in the following subsections.

3.4.1 Self-Describing Behavior. The fact that a sensor must offer some sort of self-describing behavior implies it must be an active device in terms of software, which may be rather different than what one would expect it to be. For example, a sensor may be actively generating pulses and detecting changes based on differing return signals, however, this is still passive in terms of the host software. This presents a considerable challenge to achieving a PnP sensor environment. Short of IEEE 1451, there really isn’t a standard for how a sensor describes itself to a particular software application unless it was designed ahead of time with that capability in mind. None of the sensors used in the ANT center are IEEE 1451 compliant, nor are most sensors used for navigation and positioning or robotics for that matter. This isn't as big of an issue with "smart sensors" or sensors having their own microcontrollers or FPGAs since they can simply be programmed to provide such self-describing information. Unfortunately, this isn't so easy for most sensors used in the field of interest. The Air Force Research Laboratory (AFRL) has taken the approach of adding Appliqu Sensor Interface Modules (ASIMs) to their design which resolves such problems in their PnP spacecraft efforts. Dr. Peter Wegner describes the ASIM as "a smart interface chip...analogous to a USB interface chip in a personal computer [48]. Without such workarounds available for this particular study, each sensor is treated passively; the host OS must be used to actively monitor for hardware changes.

Coupled with the need to resolve a sensor's physical location in terms of the hardware ports, this particular challenge is one that requires much more extensive resources, perhaps even a technology trend change, in order to be fully universal. As a result, the testing performed in this effort relies heavily upon hard-coded information obtained upfront. This is something that will come into play later on as well but begins due to the lack of self-describing behavior associated with the sensors being used. In these tests, the sensors that were listed previously were known as well as their interfaces, prior to attaching them to the hardware system.

3.4.2 Sensor Location. The sensors are interfaced with the host system using the available hardware ports on the PowerBot. In Ubuntu, these are mounted in different ways according to the available ports and the type of interface connection. For example, there is only one Ethernet port available, however, because it is a network interface the network address (IP) of the attached device must be known in order to use it. For the Prosilica, this wasn't too difficult to acquire; the camera can be polled directly for this using a few simple commands in ROS. The simplicity of this is possible due to the fact that the camera itself is capable of storing a specific address which doesn't change without direct and deliberate intervention by the user.

For the Serial/USB devices, there is no such address storing capability. Furthermore, since the device is connected via a serial-to-USB connection, there is no information that is automatically received by the host OS regarding the serial device itself. What can be found instead is information on the converter hardware. This is extremely limiting and stems directly from the lack of inherent self-describing behavior discussed in the previous subsection. One thing that was noticed due to trial and error was that the SICK, which is attached to the PowerBot and connected to the motherboard with the aid of a dedicated converter board within the robot. This is always mapped to /dev/ttyUSB0 on the host OS. For the Microstrain AHRS, the external serial-to-USB converter almost always mapped to /dev/ttyUSB3, however, this can be verified by observing the OS log files when the device is being mounted. This knowledge was used in writing the scripts responsible for configuration and running the sensor drivers.

To note, the Microstrain AHRS would have been connected directly to a serial port if there were one available on the PowerBot. This would have presented its own unique challenges being that no information could be detected passively. With the USB-to-serial converters, a hardware change is at least detected by the host OS which makes configuration somewhat simple leveraging the upfront knowledge previously discussed. However, with a pure serial connection, each available serial port would require active polling in order to detect when a new device has been added or removed from the system.

3.4.3 Sensor Configuration. With the aid of the ROS drivers that have already been created by the open source community, this step can be greatly simplified. The drivers for each of the sensors must be installed on the system and present in the ROS path, however, prior to using them. This obviously requires knowledge from the previous two steps since one must know the sensor being used in order to obtain the driver in the first place. The path is set ahead of time so that all newly acquired packages should be installed in a location that ROS can find. With the knowledge of the device being connected, the location where it will be mounted, and the driver located in the ROS path, configuring the sensor is now achievable either through a command terminal or bash script. As such, a script was developed for each of the sensors that runs autonomously once the sensor is connected to the system. This is discussed further in a later section.

3.5 Software Modeling

It is logical to look at virtually anything that is comprised of interacting parts as a component based system. The reader is able to find a brief explanation of this type of modeling in the CBSE section of Chapter II. To begin developing the software that will comprise the PnP architecture, the classic UML approach of generating use cases was followed. After describing the system in terms of the use cases, the overall system was then captured into a component model. This was the first attempt at describing the system using CBSE. One of the major software development artifacts of any project is the use case model and use cases themselves. This research is no exception and though this view may represent a somewhat simplistic overall interaction paradigm, it provides the foundation upon which the development can be expanded later on. The modeling of the system represents a significant portion of the methodology so an extensive explanation is warranted here. The major use cases, as well as the use case model, are included to aid in the discussion. The rest of the use cases can be inferred from the diagrams and subsequent discussion, therefore listing every one does not add any value to the reader. As such, they have been excluded in the interest of brevity but are included as an attachment in Appendix B beginning on page 114.

3.5.1 Component-Driven Use Case Model. The use case model used with a component-driven design in mind follows in Figure III.3. One thing that stands out and should be completely obvious is how one of the actors, the Host OS, has a connection to nearly all of the use cases. That is, the Host OS is a stakeholder in all of these use cases. The reason for this is the operation system must be involved in every system-level process in the software. Since this is a PnP research problem, it is only natural that most of these use cases have system-level dependencies. The “Driver Handler” actor is analogous to an aware object, or Observer from the design pattern viewpoint. The Host OS doesn’t need to know how data is being published or subscribed to as these represent user tasks. This is why there are no connections between the Host OS actor and the “Publish Sensor Data” or “Subscribe to Sensor Data” use cases. The KF Nav actor represents the portion of software developed to utilize the sensors currently provided by the PnP framework: tasks that are captured within the “Update KF Algorithm” use case. The Host OS isn’t a stakeholder for this either since it largely falls within the user layer.

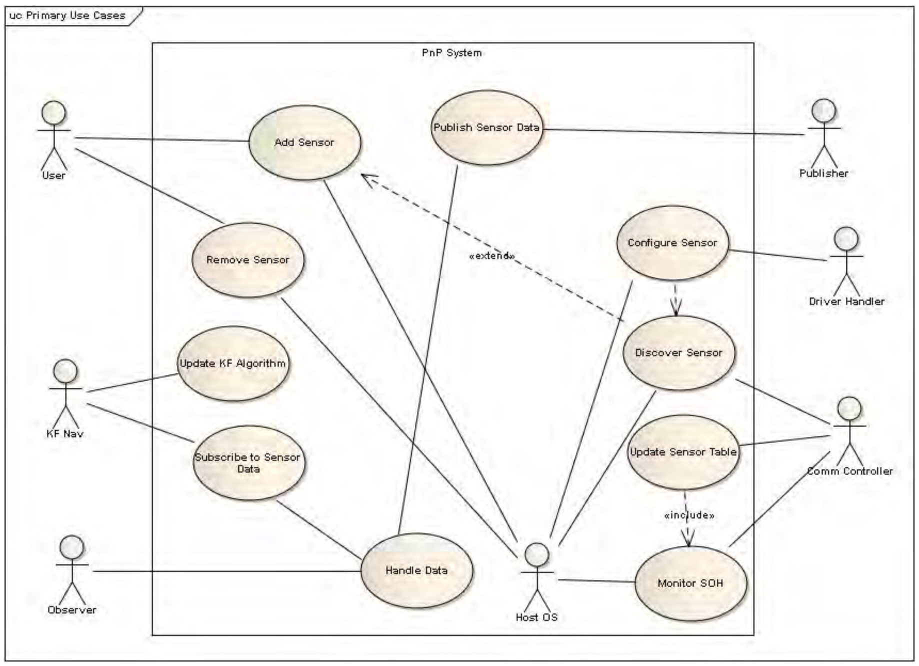


Figure III.3: Primary use cases for component-based system

The rest of the diagram will be explained much better by delving into some of the use cases themselves. To begin with, adding a sensor and removing a sensor are very similar so only one of these is included; it should be obvious from the one how the other is captured. Also, only the first use case is presented in the entire fully-dressed form. The others only show the main-success scenario (sometimes referred to as “the happy path” and extensions to that scenario which are conceivable deviations that may be encountered. Presenting the use cases in this way should capture the essence of the model without being overly detailed. The intent of showing one of the use cases in its entirety is two-fold: to show what has been left out and to assist the reader in interpreting the more complete versions included in the appendix. The numbering of the use cases is arbitrary as they all have names.

Use Case 1: Add Sensor

Primary Actor: User
Supporting Actors: Host OS
Other Stakeholders and Interests:
Comm Controller - Will need to discover the new sensor
Driver Controller - Will need to configure the new sensor for use
Pre-Conditions:
Host OS must be booted and running on the system.
Success End Condition:
Sensor is powered and/or plugged in to the system via its native interface.
Failure End Condition:
Device doesn't power or interface or startup script fails
Minimal Guarantee:
None. User must troubleshoot hardware malfunctions.
Trigger:
A user desires to use a sensor not currently connected to the system
Main Success Scenario:
1. User attaches sensor to the system using the default interface
2. Sensor is powered
3. Host OS mounts the sensor to the system bus

Extensions:
1a. In step 1, the user cannot connect the sensor
    1. User checks to see if interface connection is possible
    2. User retries or aborts depending on 1a.1.

2a. In step 2, the sensor doesn't power up
    1. User troubleshoots power connection

3a. In step 3, the sensor isn't mounted
    1. User checks whether or not the port is active and properly configured

Variations:
2'. Sensor is powered ahead of time by user with a dedicated power source
Frequency: As needed according to user requirements
Assumptions:
Host OS is capable of detecting hardware changes and running scripts

This use case is pretty self-explanatory in terms of the desired outcome. It is desired that a user can plug in a sensor, the operating system will recognize it, and mount the sensor to the system bus through use of a script. In this use case, other areas are included that add context to the scenario, such as who the main and supporting actors are, who the stakeholders are, what constitutes success, etc. With that in mind, most of the pertinent information in terms of activity are captured by the main and exception paths. Any other additional information can be added in textual explanation of the use case. This is how the rest of the use cases will be addressed. For example, it is important to know this use case will be required any time a sensor is added to the system. However, mentioning this is redundant as it is captured by the frequency field in the complete use case already provided.

Several things must happen after plugging in the sensor. For one, the sensor must be interfaced in a manner consistent with that intended by the manufacturer. Secondly, the port where the sensor is connected must be configured to accept the sensor and allow it to communicate with the system in the manner intended. Third, there must be a way to identify the sensor and track its physical location. Knowing this allows a script to configure the sensor for use and for other software to communicate with it, which is something that will be required in other use cases.

This scenario, and all of the use cases assume the user is moderately familiar with the host OS in general and knows how to troubleshoot port configuration issues. Also, it is assumed the host OS can detect hardware changes. For removing a sensor, this represents the only real requirement. A user will unplug the sensor and the operation system will no longer associate it with a port. Discovering a sensor is included in the previous use case and is completely dependent upon the host OS; it will not be discussed any further in this section.

Use Case 4: Configure Sensor

Main Success Scenario:
1. Driver handler is notified by comm controller of hardware change
2. Driver handler requests name and location from comm controller
3. Driver handler executes driver for given sensor
4. Sensor begins generating data according to launch / configuration file or other software

Extensions:

3a. In step 3, the launch file does not exist
   1. Exception is thrown and passed to comm controller
   2. User retries or aborts depending on 1a.1.

4a. No data is collected
   1. Default or blank data is present

In Use Case 4, the driver handler finds and executes the driver, readying the sensor for data collection. In order to do this, it must be notified of where the sensor is located and what type of sensor it is. This is provided here by the Comm Controller which receives the information from the host OS. The reason the driver handler requests name and location instead of the information being provided automatically is that there are other hardware changes the host OS is tracking that may not apply to a sensor change. For example, a local disc may be mounted such as a thumb drive and this has no relevance to the host OS whatsoever.

In order to use passive sensor devices, all of the drivers for any possible sensor must be present and ready for use prior to the sensor being connected. A configuration file, which basically tells the OS how to use the sensor, must also be available or written upfront. For a ROS-based system, such as the one indicated in Use Case 4, the launch file acts as the configuration file. Once the sensor is mounted and configured for use, it will begin collecting data. If no data is being collected, it will be reflected by the default or error state of the sensing device.

Use Case 5: Monitor Sensor SOH

Main Success Scenario:
1. Comm controller checks log files for output and error file messages
2. Comm controller resolves state of sensor, time stamp, and errors
3. Comm controller logs SOH data

Extensions:

1a. No error or output messages are found
   1. Comm controller logs sensor as offline

In Use Case 5, the Comm Controller is acting as an Aware object, constantly polling the data from the sensors and parsing system log files for errors pertaining to the connection with the sensor. If the sensor is taken offline, this is reflected in the log files and the Comm Controller can then update the sensors it is tracking. Checking for and time stamping the data received serves multiple purposes. For one, it is used later in the sensor table which allows other software entities to check which sensors are online and transmitting data, and when those sensors were last available. Perhaps even more obvious is that if a sensor isn't transmitting data, whether it is has been removed from the system or not, is not usable and should therefore be taken offline in terms of SOH tracking.

Use Case 6: Update Sensor Table

Main Success Scenario:
1. << *includes* >> Monitor SOH
2. Comm controller parses SOH data
3. Comm controller formats data for inclusion in sensor table
4. Comm controller adds data to sensor table

Extensions:

2a. No SOH data is available
   1. Do nothing

Use Case 6 is an includes use case meaning the "Monitor Sensor SOH" use case could be combined with with it. They are listed separately, however, since it is useful to look at these as two separate requirements. The reason for this is that the Comm Controller will log the SOH data, which can be post processed for error checking or troubleshooting purposes, then uses that data to keep an active, real-time table of available sensors. If no new SOH data has been found, the table must already reflect the most recent information and therefore doesn't need to be changed. In a ROS-based system, Use Case 5 and Use Case 6 are handled by *rostopic*.

Use Case 7: Publish Sensor Data

Main Success Scenario:
1. Publisher publishes continuously, reflecting sensor measurements
2. Streamed data is collected and handled by Observer

Extensions:

1a. No data is published
   1. Exception is thrown and sensor is ignored

2a. Null or erroneous data published
   1. Observer handles data as normal

At this point, it is beneficial to take a step backward and reexamine the experiment's overall desired outcome. It is to have the ability to plug in a sensor and for it to work autonomously. With what has been presented so far in terms of the use cases, this requires a lot of interrelated things going on behind the scenes. The sensor needs to be recognized by the operating system, mounted to a port, configured and made usable with specific drivers, after all of which the sensor can be used to obtain data. This is the first step where the data is actually being handled in some way. The reason it is the driver handler that first touches this is so ROS can be fully leveraged in terms of what it has to offer. This ties directly into step one of Use Case 7, where a launch file can be written to create specific sensor nodes that will then be tapped for the data. Since a publish/subscribe architecture is being used, a publisher (the Publisher actor in terms of the use cases) is another part of the software that will publish data streams from the sensors. Once the data is being generated by the sensor, one or several publishers will publish data directly to a Mediator. The Mediator is simply an aware object that handles data requests, both from publishers and subscribers.

For the exception path, there can only be one instance of a publisher running at a given time. If the Publisher cannot be initialized, and it isn't already running, the entire system is broken and will not provide a solution to the subject problem. Also, if the sensor isn't working correctly or is providing junk data, then whatever represents the default stream for null measurements will be passed to the Mediator. This is consistent with what is presented earlier with regard to the sensor SOH since the sensor is still connected to the system even though it may not be actively generating valid data.

Ultimately, it is up to the subscriber (KF Nav in this model) to determine whether or not to use the data based on the SOH information provided by the Comm Controller and the raw data received. Error checking could be implemented in the Mediator but that would be at the expense of reusability since each sensor will have different types of data streams. Assuming the Mediator were tasked with error checking, it would require knowledge about what any possible data stream should look like. It makes more sense for this to take place on the application side; for example, N&P applications expect imu data streams from a particular sensor in a specific format and within a certain range.

Use Case 8: Handle Data

Main Success Scenario:

1. Publisher requests to publish data to Mediator
2. Mediator accepts request from Publisher
3. Mediator adds new data channel and saves data to file
4. Mediator handles data subscription requests
5. Mediator executes data subscription requests

Extensions:

1a. The Mediator never receives a publishing request
  1. Mediator continues in current state

2a. Mediator unable to service Publisher request
  1. Publisher sleeps and retries after a specified time
  2. After a certain number of attempts, output error and give up

3a. Mediator is unable to open data channel or save data to file
  1. Throw exception and halt program

4a. Mediator is unable to handle data request
  1. Throw exception and notify subscriber

Use Case 8 is pretty self-explanatory but warrants a few explanations. First, the Mediator handles all data streams for all sensors and must do something with the data, even if it has received no subscription requests. Saving it to a file is logical as it allows the data to be examined even after an application has been terminated. Secondly, no application or software entity will have access to the data without subscribing to it. Once the publisher software has been initialized and begins running, it exists and is active until the application has been terminated. As long as the publisher is active, the mediator will continue to receive data from it. The mediator will handle the data, which may consist of passing it on to a subscriber, saving it to a file, or simply dumping / overwriting it each time new data come in.

In the exception path, the Mediator may be tied up with another process or be receiving multiple publishing requests at once. As such, if the Publisher is unsuccessful at getting a response from the Mediator it should wait for a period of time and then reattempt. After a certain number of attempts, there is likely something else wrong in which case the publishing request won't ever be serviced. It is important to note that an inability for the Mediator to open a new data stream or save data to a file represents a critical error that warrants halting the entire program. An inability to service a subscription request, however, could be the result of several factors so throwing an exception that will notify a subscriber of the problem should be sufficient.

Use Case 9: Subscribe to Sensor Data

Main Success Scenario:

1. KF Nav subscribes to a data output stream for a particular sensor from Mediator
2. KF Nav receives the measurements from the sensor from Mediator

Extensions:

1a. KF Nav is unable to subscribe to the output stream

   1. Verify request is consistent with Mediator interface
   2. Reattempt to subscribe to the data stream

The subscription use case completes the publish/subscribe pairing. There really isn't much to this; it involves requesting a data stream according to an agreed upon interface with the Mediator. The exception path is also simple since a failure to receive a data stream represents an error in the message-passing scheme. One possible source of this is the subscriber requesting data in a manner inconsistent with the given interface. Otherwise, there is some other problem either with the software linking or the Publisher itself which will require refactoring.

3.5.2 Component Model. Using the requirements and specifications brought forward with the use cases allows the system to be modeled in the form of a component diagram. Figure III.4 on page 58 shows a representation of such a PnP system with all of the required entities modeled as components. It is important to recognize the system itself can also be considered a component and also that other components could be broken down into smaller sub-components. Therefore, in determining what components to model, it's key to choose components that aren't so large they lack in providing details while not so small they are lacking in providing a coherent view of the overall system.

Looking to the diagram and referencing the use cases, "Add Sensor" and "Remove Sensor" are modeled as interfaces that trigger the changes in the PnP system. All of the actors are represented as components in this diagram, along with Nav Sensor Table. Of course the sensor itself is looked at as a component, and to note, it represents one that cannot be broken down into smaller sub-components. The Comm Controller and Driver Handler components can be grouped together as message-oriented middleware and is where much of the system-level and application work will be focused. The functionality of the Driver Handler is largely provided for by the implementation; namely, the specific sensor drivers themselves. Similarly, the Comm Controller is very dependent upon system-level tools that can supplement and gather information from the Host OS. The

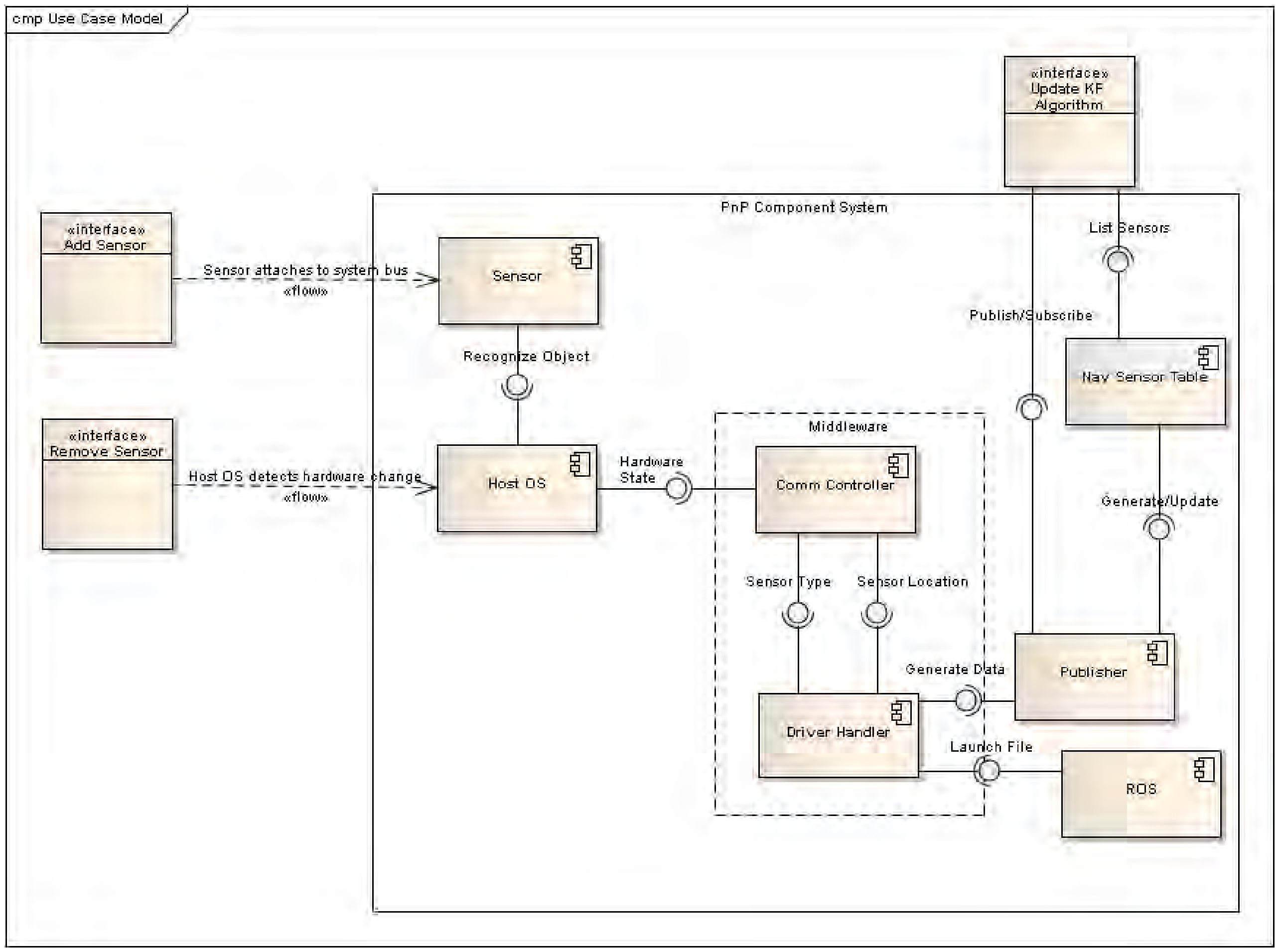


Figure III.4: Possible component diagram of PnP sensor system

sensor table is generated by the Observer and KF Nav (subscriber) will use this to request data from the Observer directly.

All of the components within the boundary represent the complete PnP system which is why “Update KF Algorithm” is listed outside the boundary. Of course it really wouldn’t make much sense in practice to publish data and for it not to be used. This aspect of the data collection is really outside the scope of what is to be accomplished, however, since the PnP solution is achieved once the data can be published. That is, the Publisher will keep track of all sensor data streams and can simply wait for a request before publishing it. What data streams are used and for what purpose is a separate consideration outside of the system and is application dependent.

3.5.3 Modeling a Component-Based System using ROS. Now that a component-based system has been modeled, ROS can be used as middleware to account for much of the functionality previously described. Namely, in a ROS-centered system one use case, call it "Handle Sensors," would include all of the required drivers, data handling, and communication required. It also keeps track of all of the sensors currently online and available to a subscriber. This is reflected in the modified use case diagram in Figure III.5.

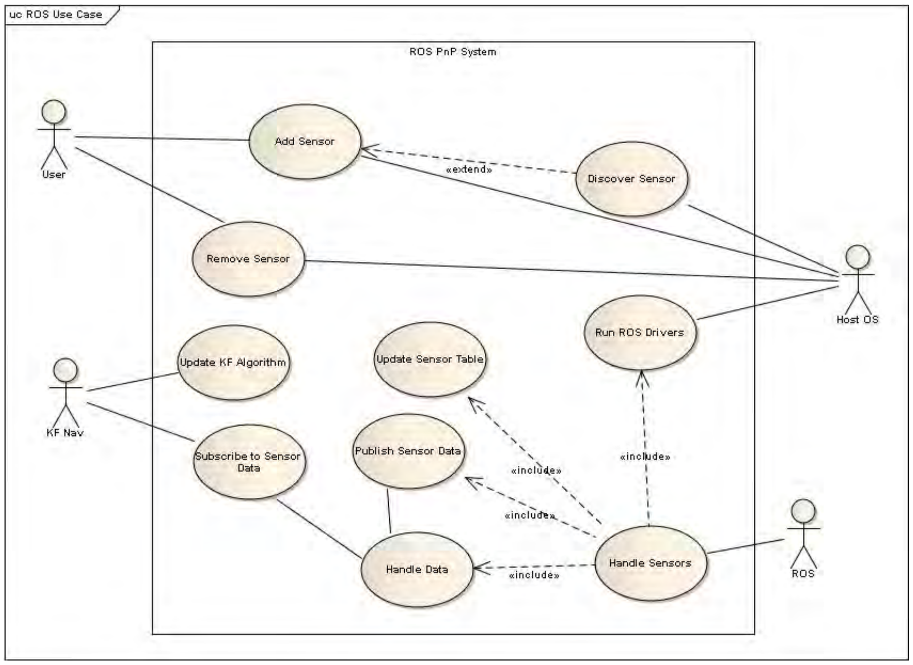


Figure III.5: ROS-centered use case diagram for a PnP sensor system

There isn't much need to show the written use cases for this modified system as the aim isn't altered in any way. ROS simply provides much of the functionality required by the PnP system inherently by design. Of course, looking at the software entities as components, the component model will change as well. A component diagram for the

ROS-based PnP system can be found in Figure III.6. Again, there isn't much need to go into detail describing the model. It is virtually the same as that found in Figure III.4 on page 58 except ROS replaces many of those components. This is a result of the middleware itself being viewed as a component encompassing their functionality.

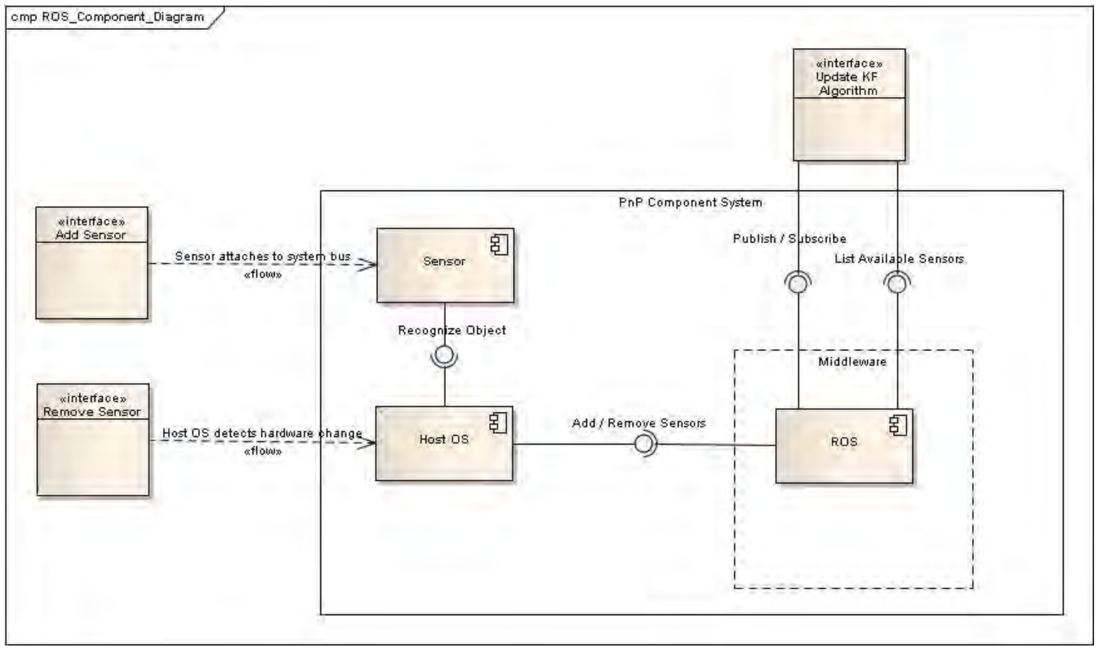

cmp ROS_Component_Diagram
«interface» Update KF Algorithm
PnP Component System
«interface» Add Sensor
Sensor attaches to system bus «flow»
Sensor
Publish / Subscribe
List Available Sensors
Recognize Object
Middleware
«interface» Remove Sensor
Host OS detects hardware change «flow»
Host OS
Add / Remove Sensors
ROS

Figure III.6: ROS-centered component diagram for a PnP sensor system

3.6 Autonomous Behavior Using Scripting

Much of the behavior expected in a PnP system when a sensor is added or removed from the system should be fully automated. In computing, scripts are used to achieve this sort of behavior for many different applications. For example, when a computer is booted and the operating system loads, there are many different scripts running in the background setting up the session for a user in a specific configuration. Similarly when hardware changes take effect, scripts are often used to configure and find drivers for those devices autonomously so they can be used right away. The same idea applies here, however, the scripts must be specific to achieve the results desired for sensor PnP.

As was mentioned previously, the operating system used for this work and all subtasks is Ubuntu Linux. This is fortunate as scripting can be done very easily within a Linux environment. For one, anything that can be input into a command line and executed by a user can be written into a script and executed with the same effect. Not all scripts are created equal and there are even different kinds of scripts. For example, in Windows the primary type of scripting is what is known as "batch programming" (DOS scripting) while in the Linux world it is "shell scripting. There are some differences, however, the end result is virtually the same. A short review of the differences could be made, but, it doesn't really add any value to this section. Another thing to note is that on the Linux side, some scripts will be referred to as bash scripts and some as shell scripts. For the purposes of this paper, they can be looked at as being interchangeable.

3.6.1 udev. ArchLinux and Wikipedia define udev as "the device manager for the Linux kernel. Primarily, it manages device nodes in /dev. It is the successor of devfs and hotplug, which means that it handles the /dev directory and all user space actions when adding/removing devices, including firmware load. ArchLinux further asserts that udev encompasses the functionality of both hotplug and hwdetect [1]. Not all that long ago udev didn't exist and thus /dev was much less dynamic. The environment proved undesirable for many users so changes were made incrementally before udev eventually came into being. Now, udev can be used to detect hardware changes, create nodes for the new hardware in /dev, and manage permissions [38].

There is much literature and material freely available on the web that describes udev in much greater detail, as well as examples on how to begin using it in an application. In fact, the internet, especially Linux-specific forums, represents the authority when it comes to information pertaining to udev. This makes sense due to its nature and the same can be said for most open source software. Knowing what has been illustrated thus far in terms of capabilities and requirements, udev fits nicely into the realm of solving the stated research problem.

This is precisely the approach taken, with udev rules being coupled with scripting to resolve the autonomous requirements for the desired PnP system. There is much literature in the Linux community regarding how to write udev rules to provide autonomous and consistent behavior once a USB device is installed in a system. These forums, blogs, wikis, and articles represent the core of the knowledge-base for this subject as is the case with other Free and Open Source Software (FOSS). This is only natural as the software is developed, maintained, and revised by the community of users. All of the information for setting up and using udev for this application was obtained through such sources and are referenced accordingly.

Before discussing the specifics of using udev, it is necessary to understand in some degree what it actually does. That is, the question should be answered as to how udev allows autonomous behavior when a device is mounted to the system. For one, udev is dependent upon the interaction or communication between the udev configured rules and sysfs. sysfs is native to the Linux 2.6 kernel and represents a means of transporting information from the kernel code to user processes. In other words, it is "a mechanism for representing kernel objects, their attributes, and their relationships with each other [27]. When is mounted to the system, udev will match its rules against the information it receives about a device from sysfs in order to recognize it. Rules can also include other information for udev specifying what should be done once a match has been made, such as creating a symlink (symbolic link) or running additional programs, such as device drivers [22].

The Ubuntu Manpage serves as a very good reference manual for udev. There are many pages that can be found which outline step-by-step how to use udev for a particular application, such as automatically backing up certain files to a USB drive once it has been plugged in. However, when attempting a custom application someone else may not have covered in a post, one will require an understanding of what udev provides, its limitations, and overall syntax. For example, all of the field names that can be used to check for device properties is listed in the manpage. This is vital knowledge for someone writing a rule that loads the drivers for a particular device whenever it is attached to the system.

There aren't any tutorials or articles that could be found describing exactly how to recognize a sensor and run the drivers for it once it has been mounted. Based on what is required for this part of the problem, some of the common applications udev has been used for can be leveraged to aid in realizing a solution. Take the hard drive backup example mentioned previously. For this to work, udev must run a script once the device is recognized that will access a folder, check for file changes, copy new files and save them to another folder. While this isn't exactly what needs to happen with the sensor once it's mounted, some of the basics are the same. Another useful insight is that udev rules can specify a certain script (or multiple) to be run once the device is mounted. As mentioned previously, the script can contain any commands normally entered by way of terminal.

In order to use udev to perform user-specified rules, there are several things that must be known upfront. For one, the writer must know what device is being connected and where it's connected. This of course doesn't imply universality, but with a passive device, developing a solution that is would present a considerable challenge for reasons already addressed. There are software tools that can be used to check for hardware changes or one could simply monitor the system log files to gather this information. One problem with this is that there are no serial ports for the serial devices being used, requiring hardware detection tools to recognize the serial-to-usb converter rather than the serial device being connected. This is fine if one knows what converter is associated with what sensor but presents a rather interesting challenge if this is not the case. Whatever method is used to detect the hardware change will specify the location of where the device was mounted. For example, a USB device may be mounted to /dev/ttyUSB0. The name and type of device are also indicated; however, since it is often the case this is known upfront, this information may or may not be required. Also, with only the location known, udev can be used to gather a rather extensive list of information pertaining to the device mounted there.

Once the location is known, udev can be used to gather information about the device. Perhaps most significantly, a list of attributes for the device can be found which can then be used to write specific udev rules. The attributes help ensure the rules are written for a specific device. This is realized by the manner in which udev compares its rules against any device introduced into the system at a given location. Another feature that makes udev more adaptive than what may be apparent is through the use of what is referred to as a wild card. These wild cards work based on the knowledge of what type of device is being connected and the naming convention inherent in Linux. For example, an external hard drive might normally be mounted by default at /dev/sda1. Assuming there is already a device mounted at that location, the system may then mount the drive to /dev/sdb1, and so forth. Knowing that changes are reflected in the third character of the lowest-level folder name, a wild card can be used be inserting the question mark character. That is, for a hard drive the rule can be written with an effective location of /dev/sd?1 [43]. Finally, the rule can be used to run scripts or any other program to effect the desired autonomous behavior.

3.7 Publishing Data

Once the system has configured the new sensors and run the drivers to use it, collecting data can be accomplished rather easily. One way to do this is manually, with the user controlling when to activate the sensor, the tools used to access the data being collected, deciding what to do with the data after it's collected, etc. Since this research is focused on a system of sensors requiring little to no outside intervention, this approach cannot be taken directly. However, with the advent of the system tools that will be employed (udev, etc.), gathering the data can be accomplished even faster and easier. Each sensor can be set up to broadcast continually or at specific intervals as long as it is available; the data can be output to a stream, saved to a file, ignored, etc. The determination on what to do with the data is a responsibility of the Mediator. The caveat to this is, whatever the Mediator does with the data, it must always be capable of handling any Subscriber request.

For this system, each sensor will begin collecting raw data as soon as it's available which in turn will be passed to the Mediator. There are no considerations made for what data may be needed or how it will be used, meaning it is passed to the Mediator and whatever is done with it from that point is not relevant to the Publisher. The Mediator, therefore, has access to the must up-to-date data at all times and can use it to service subscription requests. This approach allows for more flexibility in terms of applications that can use the underlying PnP sensor solution. By decoupling the functionality of the discovery and configuration components from the application-specific software, a user will always have full access to any of the data being collected from the suite of sensors. In essence, the PnP software can be used as a COTS component and will interface directly to it using publish/subscribe. Once this is done, the user software can request a list of available sensors and subscribe to any or all of the data streams.

Not only may it be necessary to store the data collected to a file to fully service a subscription request, it is also helpful to have in case it is needed later on. Specifically, the data can be used in the same manner it has in the past for post-processing and error resolution. Furthermore, the handling of data is contingent upon the sensors being active. This will require a means for tracking what sensors are currently available, such as a sensor table. The system can be more robust by also tracking SOH data utilizing real-time error checking but this isn't required in order to log and transmit the data. In other words, if the data is available it can be used, regardless of whether or not it is accurate. There is utility in this as well; even inaccurate data can be incorporated into a Kalman filter, providing there is some sense of how inaccurate it actually is. Also, the logging of such data can be used later for trend analyses and other such advanced research pertaining to a particular sensor's performance.

3.8 Handling Data

For this work, it is ideal that all requests for data requests will be handled through a dedicated Mediator. Naturally, the GoF Mediator pattern is a very good choice in attempting to accomplish this. This is consistent with the publish-subscribe paradigm with one entity or section of software handling all of the data to include both publishers and subscribers. This also parallels the strategy introduced by Fortier, which is discussed in Section 2.10 of Chapter II, in which he proposes the use of the Observer pattern to model the aware objects, allowing them to receive information about context changes [10]. A Mediator pattern is very similar to the Observer pattern; in fact, the Observer pattern can actually be used to implement the Mediator pattern [11].

The Observer pattern is one way to model the publish/subscribe paradigm mentioned frequently within this paper. It is used to handle situations where there are many subscribers, or observers, communicating with one (or few) publishers, or subjects. The Mediator pattern can be viewed as an extension of this to handle the case where there are many publishers and many subscribers. The mediator entity thus provides a one-to-many communication scenario in terms of interacting with both the publishers and the subscribers —ıthat is, all publishers and subscribers communicate with the mediator directly. The mediator is the sole entity with exclusive responsibility for all data handling. The users of the data will represent the subscribers, such as KF Nav in the component model, and the data streams are the subjects, or publishers.

The reason a pattern such as the Observer or Mediator are even needed is to allow many applications to use the data provided by the sensors without the added complexity inherent of multiple, and often differing requests. More specifically, the mediator needn't be concerned about tailoring what data it's broadcasting to what, or how many, subscribers. It can simply broadcast all of the data and it is up to the subscriber to determine which subjects, or data streams, to subscribe to. Using such an approach in the design for this particular PnP framework portends scalability and portability of the completed software, allowing for a more robust advanced navigation solution as well as the capability of its use in a completely different application. For the immediate need of direct use in a particular navigation and positioning application, the pattern really isn't necessary. The reason for this is there is really only one subscriber, so there is already one-to-many communication between subscriber and publishers from the onset.

The approach taken in publishing the data takes both immediate and future considerations into account. The first approach leverages ROS, incorporating into the model as middleware. In this approach, ROS is active in initializing the sensors, handling their data, and keeping track of what sensors are connected to the system at all times. The data publishing is provided by ROS and subscribed to by the application software. A more traditional communication approach is also considered where each data stream is treated as a publisher and the navigation application as the subscriber. Conducting the tests in this manner allows the initial focus to be placed on the immediate need while providing a platform for comparison between the two approaches. It also seeks to establish a foundation for future enhancements through the use of well-designed and accepted patterns.

3.9 Summary

The software unit to be developed ends with the realization of the functionality described in the previous sections. That is, it presents an environment which recognizes when a sensor is added or removed from the system and handles the requests for its data accordingly. In order for this to happen, each sensor must be mounted to the system bus, the drivers must be executed so the sensor will operate correctly, and its data must be handled in a way that allows an application to access it. Furthermore, there must exist a method of detecting the sensor SOH so that only valid, functioning sensors will be used by the subscribing application. One such example of this is when a sensor is disconnected. In this case, there should be no scenario whereby an application will attempt to receive data from it. If the sensor isn't currently subscribed to by an application, it will simply cease to appear as a choice. Otherwise, if an application is currently subscribed to the data stream of a particular sensor when it goes offline, the application must be made aware of this in some way so it will stop attempting to use it.

IV Results & Analysis

The experiments and analysis for adaptive PnP can be broken into four main sections: 1) Automated Behavior, 2) Data Handling, 3) Data Use, and 4) Comparisons to Traditional Approach. For this chapter, the main comparisons are centered around ROS and how it aids in providing much of what is required for a PnP sensor solution in contrast to what is currently employed, either in a component-based or object-oriented approach. This involves the study of how ROS can be employed, how sensor data is currently utilized, effectiveness of CBSE and design patterns and their limitations, and various combinations that may afford the best solution to a particular navigation problem.

The work conducted can be viewed as a proof of concept or feasibility study for sensor PnP. As this is very much an engineering focused thesis, much of the work depends upon what tools are currently available. As the nature of the problem has imposed the use of Ubuntu, ROS and other FOSS software, the knowledge base is mostly limited quite to user forums. Much of the information gathered, to include code snippets and full programs, have been taken from these sources and will be referenced accordingly. As is stated in the previous chapter, there are no sources more credible in relation to this type of software: it was developed by the very community supporting it. In fact, the interaction on such mediums (forums, blogs, lists) takes software evolution full circle with what has come to be known as "design by blog," wherein feedback from the users of FOSS drives future development [45]. The changes can be either direct, where a user makes the change himself, or indirect meaning developers make changes based on the insights of others. The downside to this type of development is that, without a governing body or version control, changes in packages and extensions my cause conflicts with other software components.

The results of each section will be addressed individually and summarized collectively at the end of the chapter. Code snippets and diagrams are used to describe the relevant aspects of a PnP system, however, working results of code execution is limited as this is a textual artifact. Other limitations and assumptions that have not been previously introduced will be explained as necessary. For the Data Use section, examples will be made of how a user could take data from the PnP system and use it for a positioning problem, however, one must bear in mind a working PnP system is application independent. For this reason, the system is complete once it is capable of accommodating hardware changes and handling service requests for each available sensor.

4.1 Automated Behavior Results

In order to achieve the desired behavior, shell scripts were written and executed for each of the sensors. Basically any behavior that is achievable manually through a command prompt can be realized through scripting. This is precisely what was done for the Microstrain AHRS and Prosilica camera. The approach taken was to learn how these sensors are mounted, configured, and how to gather data from them interactively first and then automate the process using this knowledge. A lack of expertise in the subject of scripting imposed considerable spin-up time in this area. Additionally, to save time and decrease complexity, the straightforward approach of writing a script that mimics the effect of interacting through a terminal was chosen in all cases where a script was needed. Therefore, the scripts that were written are easy to understand and intuitive albeit not necessarily the most eloquent or resource efficient. Examples follow that show how this was achieved for the Microstrain AHRS.

4.1.1 Configuration and Startup. Before any of the data can be drawn from a sensor using ROS, the test environment must first be set up properly to allow for this. First, all of the software drivers must be located on the ROS package path in order for them to be found when they are called. Secondly, the ports must be configured to accept a new sensor and allow for commands to be executed across them. Finally, "*roscore*," which stands for "ROS core, must be running in order to use any of the sensors in the ROS environment. There are several ways to do this, however; producing an all-inclusive list would be pointless. What follows is an explanation of the selected solution along with other possibilities that may be appropriate.

Being that the sensors used for this experiment are mainly passive devices, getting any sort of self-describing behavior from the sensors autonomously presents a considerable challenge and was not achieved fully in the testing performed. Additionally, one would need some semblance of the actual sensors to be used ahead of time if they are to be used in ROS. This is a requirement not only for the custom scripts that allow autonomous execution, it would be impossible to gather the drivers needed to run those sensors in ROS without that information. Simply put, this part of PnP falls short of what the name implies. However, the tradeoff is worthwhile in the benefits that are achieved through using ROS as will be demonstrated in throughout the rest of this section. Even without true self-describing behavior, there are many tools available that help achieve the desired effect, though requiring a bit more time and effort.

The approach ultimately chosen involves polling the log files to discover what location a particular sensor was mounted to and using a combination of other software tools, such as the open source “hardware identification system” aliased as “hwinfo” to gather more information about the device when required. Returning to the AHRS for an example, the system log file was observed upon connecting / disconnecting the converter to the USB port. The relevant portions are included here for reference.

```
06:33:02 Dan-LinuxBox kernel: [ 2212.932848] usb 1-6.4.1: new high speed USB device using ehci_hcd and address 13
06:33:02 Dan-LinuxBox kernel: [ 2213.041411] hub 1-6.4.1:1.0: USB hub found
06:33:02 Dan-LinuxBox kernel: [ 2213.041589] hub 1-6.4.1:1.0: 4 ports detected
06:33:02 Dan-LinuxBox kernel: [ 2213.342743] usb 1-6.4.1.1: new full speed USB device using ehci_hcd and address 14
06:33:03 Dan-LinuxBox kernel: [ 2213.543747] usb 1-6.4.1.2: new full speed USB device using ehci_hcd and address 15
06:33:03 Dan-LinuxBox kernel: [ 2213.550666] usbcore: registered new interface driver usbserial
06:33:03 Dan-LinuxBox kernel: [ 2213.550710] USB Serial support registered for generic
06:33:03 Dan-LinuxBox kernel: [ 2213.763754] usb 1-6.4.1.3: new full speed USB device using ehci_hcd and address 16
06:33:03 Dan-LinuxBox kernel: [ 2213.982716] usb 1-6.4.1.4: new full speed USB device using ehci_hcd and address 17
06:33:03 Dan-LinuxBox kernel: [ 2214.104868] usbcore: registered new interface driver usbserial_generic
06:33:03 Dan-LinuxBox kernel: [ 2214.104876] usbserial: USB Serial Driver core
06:33:03 Dan-LinuxBox kernel: [ 2214.118337] USB Serial support registered for FTDI USB Serial Device
06:33:03 Dan-LinuxBox kernel: [ 2214.123016] ftdi_sio 1-6.4.1.4:1.0: FTDI USB Serial Device converter detected
```

```
06:33:03 Dan-LinuxBox kernel: [ 2214.123129] usb 1-6.4.1.4: Detected FT232RL
06:33:03 Dan-LinuxBox kernel: [ 2214.123134] usb 1-6.4.1.4: Number of endpoints 2
06:33:03 Dan-LinuxBox kernel: [ 2214.123139] usb 1-6.4.1.4: Endpoint 1 MaxPacketSize 64
06:33:03 Dan-LinuxBox kernel: [ 2214.123143] usb 1-6.4.1.4: Endpoint 2 MaxPacketSize 64
06:33:03 Dan-LinuxBox kernel: [ 2214.123147] usb 1-6.4.1.4: Setting MaxPacketSize 64
06:33:03 Dan-LinuxBox kernel: [ 2214.123974] usb 1-6.4.1.4: FTDI USB Serial Device converter now attached to ttyUSB3
06:33:03 Dan-LinuxBox kernel: [ 2214.125626] usbcore: registered new interface driver ftdi_sio
06:33:03 Dan-LinuxBox kernel: [ 2214.125638] ftdi_sio: v1.6.0:USB FTDI Serial Converters Driver
06:33:03 Dan-LinuxBox modem-manager: (ttyUSB3) opening serial device...
06:33:03 Dan-LinuxBox modem-manager: (ttyUSB3): probe requested by plugin 'Generic'
------------------------------------------------------------------------------------------------------------------------
06:33:51 Dan-LinuxBox kernel: [ 2261.593557] usb 1-6.4.1.4: USB disconnect, address 17
06:33:51 Dan-LinuxBox kernel: [ 2261.593699] ftdi_sio ttyUSB3: FTDI USB Serial Device converter now disconnected from ttyUSB3
06:33:51 Dan-LinuxBox kernel: [ 2261.593725] ftdi_sio 1-6.4.1.4:1.0: device disconnected
```

From the log file, it is seen where the converter device is recognized and mounted to the system. The particular location is shown as /dev/ttyUSB3, although the other three ports for the converter are also opened and mounted as well. The file also shows where the host OS locates and executes the driver so the device will work as expected. The dash line separates the log file from where the sensor is connected and where it is disconnected. As is shown, the operating system recognizes when the device has been unplugged and unmounts it from the port.

While the log file presents a good amount of information on its own, it is quite a bit limited. For example, a better name for the device and other identifying characteristics may be desired, which is not explicitly shown. This is where other tools, such as hwinfo can be of some assistance. For this example, hwinfo was used to generate information about all of the USB devices connected to the system. The following results reflect the relevant information that was found.

```
31: USB 00.0: 0700 Serial controller
  [Created at usb.122]
  UDI: /org/freedesktop/Hal/devices/usb_device_403_6001_A7006nZm_if0_serial_usb_0
  Unique ID: ApIU.OX+eFbXgyy1
  Parent ID: 7UUv.WvppkBPnhU7
  SysFS ID: /devices/pci0000:00/0000:00:0b.1/usb1/1-6/1-6.4/1-6.4.1/1-6.4.1.4/1-6.4.1.4:1.0
  SysFS BusID: 1-6.4.1.4:1.0
  Model: "Future Technology Devices International 8-bit FIFO"
  Hotplug: USB
  Vendor: usb 0x0403 "Future Technology Devices International, Ltd"
  Device: usb 0x6001 "8-bit FIFO"
  Serial ID: "A7006nZm"
  Driver: "ftdi_sio"
  Driver Modules: "ftdi_sio", "ftdi_sio"
  Device File: /dev/ttyUSB3
  Speed: 12 Mbps
  Module Alias: "usb:v0403p6001d0600dc00dsc00dp00icFFiscFFipFF"
  Driver Info #0:
    Driver Status: ftdi_sio is active
    Driver Activation Cmd: "modprobe ftdi_sio"
  Config Status: cfg=new, avail=yes, need=no, active=unknown
  Attached to: #27 (Hub)
```

Some of this information echoes what was already obtained through the syslog file, however, there are additional details. For one, the information, even that which is redundant, is presented in a much clearer manner that is easier to read and decipher than before. All of the information is presented in a specific format, so it's much easier to parse if one were to use the results as inputs to a separate program. Further, it can be seen that other information not previously obtained is shown, such as the vendor name, the model, the serial ID, device speed, etc. This information is very useful as will be demonstrated in the subsection pertaining to udev rules.

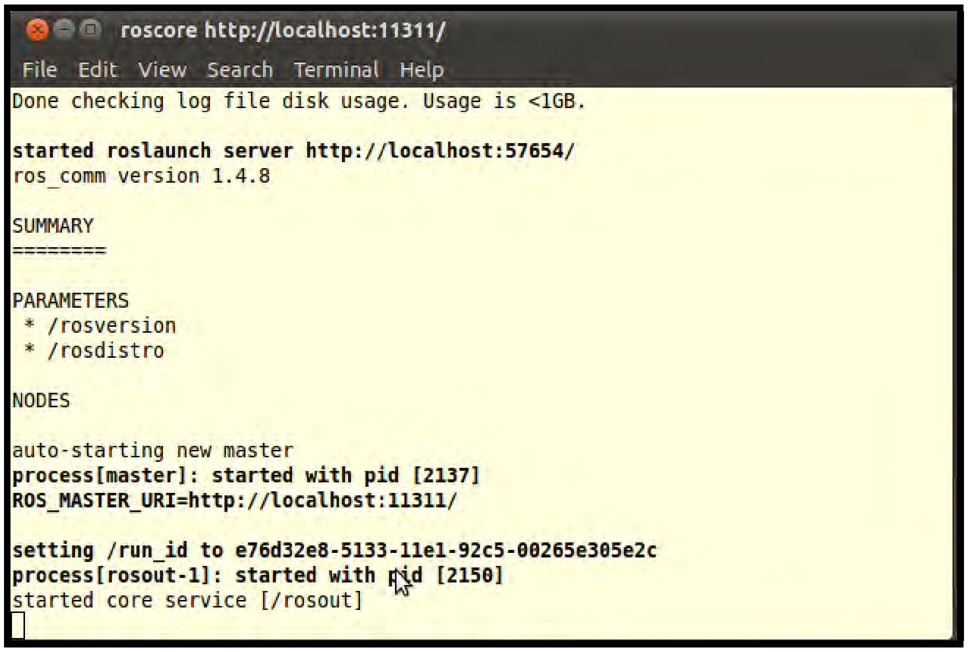
With all of the basic information known about the sensor, the drivers to use them can be acquired directly from the ROS repository. Even those that aren't available as binaries can be found and compiled from source. For all of the sensors used in this experiment, ROS drivers were already written and are readily available. This is the case for nearly all of the sensors used in navigation and positioning applications.

Once the drivers are installed and executed, the sensors begin gathering data; all of the configuration and setup has been completed. There are several commands that must be performed in order for ROS to be used to do this. This brings us back to the scripting that was introduced at the beginning of the section. For the AHRS, the stack containing the imu driver, launch files, and publishing software were acquired and added to the ROS package path. With all of the information about the sensor at hand and the software needed to control it, a script was written to automate the process of configuring and using it. Certain setup steps and tasks must be completed in order to use ROS directly as it is in the script but an explanation of this is unwarranted as anyone would require knowledge of this in order to use ROS in the first place.

```
#!/bin/sh
source /opt/ros/diamondback/setup.sh
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://localhost:11311
gnome-terminal -x bash -c 'roscore; read x'
sleep 10
gnome-terminal -x bash -c \
  'rosrun microstrain_3dmgx2_imu imu_node _port:=/dev/ttyUSB3; read x'
rosrun microstrain_3dmgx2_imu imu_node _port:=/dev/ttyUSB3
sleep 10
gnome-terminal -x bash -c 'rostopic echo imu/data; read x'
```

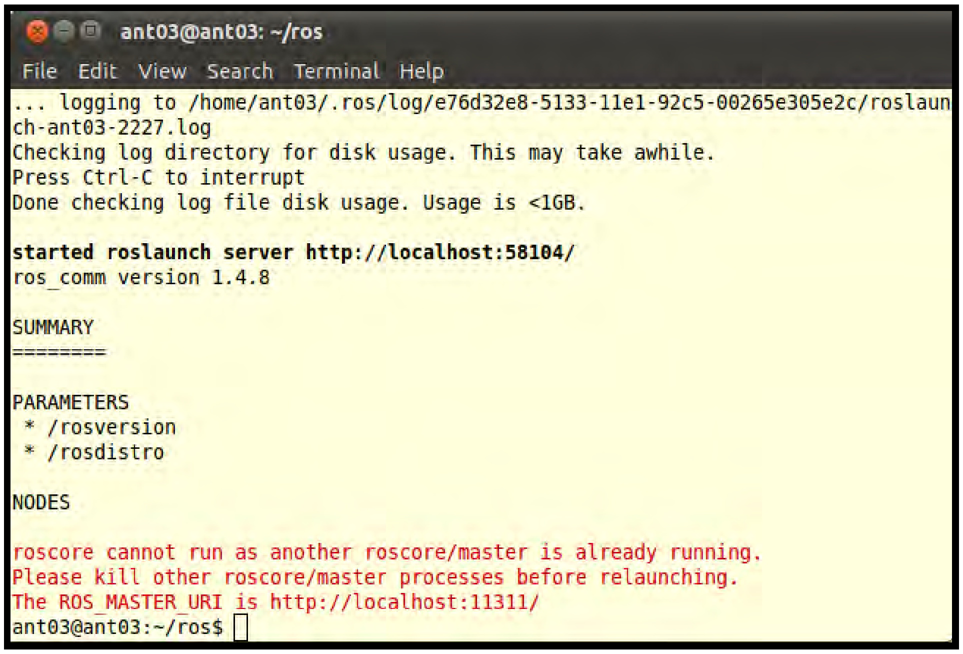
The script is executing ROS commands for the sensor, utilizing what is known about where the sensor is located. The first part is performed to specify a single host is being used. ROS can be, and often is, used across multiple machines so this would change depending on the number of hosts. Another point to note is that *roscore* only needs to be run once in order to execute the ROS commands. It is perfectly acceptable, however, to run *roscore* for each script just in case it isn't already running. If *roscore* were in fact running and then called again, it is simply ignored. An output is generated notifying the user that *roscore* is already running, but it does not interfere with any of the other commands in any way. The output viewed after running *roscore*, both fresh and after it's already running, are shown in Figures IV.1 and IV.2, respectively.

So while it may not be the most efficient way of doing things, running *roscore* at the beginning of any ROS script doesn't inhibit the ability for the rest of the script to run. This is a good fallback since it is desirable for anyone to be able to write a script that will execute desired behavior for a particular sensor. Flexibility is therefore extended by the


roscore http://localhost:11311/
File Edit View Search Terminal Help

```
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:57654/
ros_comm version 1.4.8

SUMMARY
========

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [2137]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to e76d32e8-5133-11e1-92c5-00265e305e2c
process[rosout-1]: started with pid [2150]
started core service [/rosout]
```

Figure IV.1: Terminal Output When ROS is First Run



```
... logging to /home/ant03/.ros/log/e76d32e8-5133-11e1-92c5-00265e305e2c/roslaun
ch-ant03-2227.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:58104/
ros_comm version 1.4.8

SUMMARY
========

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

roscore cannot run as another roscore/master is already running.
Please kill other roscore/master processes before relaunching.
The ROS_MASTER_URI is http://localhost:11311/
ant03@ant03:~/ros$
```

Figure IV.2: Terminal Output When *roscore* is Executed While Already Running

fact that each sensor script could be written as if it were the only sensor being used. Again, this will introduce an error, but more importantly, it will not prevent any of the sensors from working.

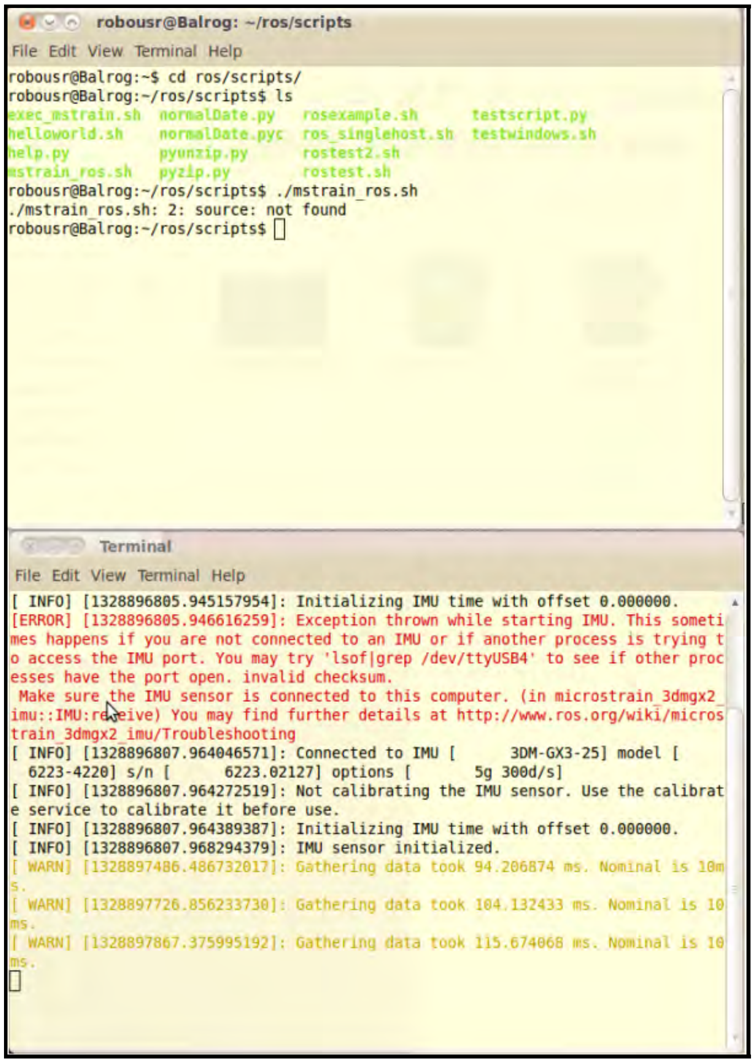
Although it has been shown that attempting to run multiple instances of *roscore* does not necessarily hurt things, it is certainly not the best of practices and can be prevented fairly easily. One method that can be used to account for this is to use conditionals based on an equality check using a regular expression (regex) and the results of the *rostopic list* command. If */rosout* is found, then *roscore* is already running, and vice-versa. Another, possibly more scalable, way to achieve the same result is to force adherence to a certain protocol, such as ensuring *roscore* is executed prior to running any of the sensor-specific scripts. This can be done with a separate configuration script or interactively through the command line. The biggest advantage to using this approach is that *roscore* isn't tied to one configuration. For example, *roscore* is usually executed with a single host, which is the default configuration. This can be enforced manually in the following manner:

```
$export ROS_HOSTNAME=localhost
$export ROS_MASTER_URI=http://localhost:11311
$roscore
```

Of course, since this is several lines to perform each time a forced single-host application is required, these lines have been added directly to a script, such as the previous AHRS example. However, some circumstances require the use of ROS across multiple hosts. For example, controlling the Powerbot using a laptop computer remotely would require two hosts. This can be performed easily enough in ROS but requires *roscore* to be configured to do so. Once again, a script is the best way to handle this.

With these concepts in mind, the outcome was verified with a simple example. Returning to the AHRS, the results of executing a script upon mounting the sensor were verified in both the single host case, and the dual-host case where the laptop was utilized as a base station. The script for the single host case was already introduced; the script for the multi-host case is virtually the same except that *roscore* must already be running on the base station prior to running the script on the host device. Therefore, the script doesn't require anything pertaining to *roscore*. The same can be true for the single host case as was mentioned before as long as it is understood that *roscore* will be running prior to connecting any of the sensors.

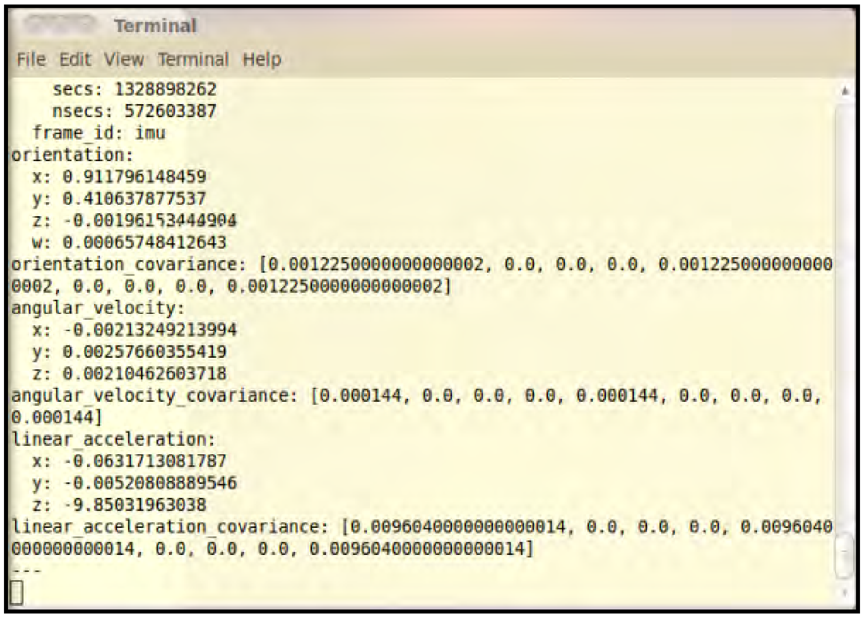
Figure IV.3 captures the results of running the AHRS scripts in both single and dual-host configurations.



```
robousr@Balrog:~$ cd ros/scripts/
robousr@Balrog:~/ros/scripts$ ls
exec_mstrain.sh  normalDate.py   rosexample.sh       testscript.py
helloworld.sh    normalDate.pyc  ros_singlehost.sh   testwindows.sh
help.py          pyunzip.py      rostest2.sh
mstrain_ros.sh   pyzip.py        rostest.sh
robousr@Balrog:~/ros/scripts$ ./mstrain_ros.sh
./mstrain_ros.sh: 2: source: not found
robousr@Balrog:~/ros/scripts$ 
```

```
[ INFO] [1328896805.945157954]: Initializing IMU time with offset 0.000000.
[ERROR] [1328896805.946616259]: Exception thrown while starting IMU. This someti
mes happens if you are not connected to an IMU or if another process is trying t
o access the IMU port. You may try 'lsof|grep /dev/ttyUSB4' to see if other proc
esses have the port open. invalid checksum.
 Make sure the IMU sensor is connected to this computer. (in microstrain_3dmgx2_
imu::IMU:receive) You may find further details at http://www.ros.org/wiki/micros
train_3dmgx2_imu/Troubleshooting
[ INFO] [1328896807.964046571]: Connected to IMU [      3DM-GX3-25] model [
  6223-4220] s/n [      6223.02127] options [       5g 300d/s]
[ INFO] [1328896807.964272519]: Not calibrating the IMU sensor. Use the calibrat
e service to calibrate it before use.
[ INFO] [1328896807.964389387]: Initializing IMU time with offset 0.000000.
[ INFO] [1328896807.968294379]: IMU sensor initialized.
[ WARN] [1328897486.486732017]: Gathering data took 94.206874 ms. Nominal is 10m
s.
[ WARN] [1328897726.856233730]: Gathering data took 104.132433 ms. Nominal is 10
ms.
[ WARN] [1328897867.375995192]: Gathering data took 115.674068 ms. Nominal is 10
ms.
```

Figure IV.3: Execution of AHRS Single Script



```
    secs: 1328898262
    nsecs: 572603387
  frame_id: imu
orientation:
  x: 0.911796148459
  y: 0.410637877537
  z: -0.00196153444904
  w: 0.00065748412643
orientation_covariance: [0.0012250000000000002, 0.0, 0.0, 0.0, 0.001225000000000
0002, 0.0, 0.0, 0.0, 0.0012250000000000002]
angular_velocity:
  x: -0.00213249213994
  y: 0.00257660355419
  z: 0.00210462603718
angular_velocity_covariance: [0.000144, 0.0, 0.0, 0.0, 0.000144, 0.0, 0.0, 0.0,
0.000144]
linear_acceleration:
  x: -0.0631713081787
  y: -0.00520808889546
  z: -9.85031963038
linear_acceleration_covariance: [0.0096040000000000014, 0.0, 0.0, 0.0, 0.0096040
000000000014, 0.0, 0.0, 0.0, 0.0096040000000000014]
---
```

Figure IV.4: Data Stream Produced using AHRS Single Script

There appears to be some timing issues producing some warnings when this is executed but the single host case definitely works. Figure IV.4 shows the data output to standard out, or the terminal screen. It should be noted that the data streams continuously while the AHRS node is operating. The reason for this is that the sensor node is publishing data as it resolves measurements, and the screen is subscribing to that data as it becomes available. Moving the sensor around caused these values to change as expected. This is more apparent using RXPlot, which is a data plotting tool native to ROS. Figure IV.5 shows the plots of two data sets. The top plot shows orientation relative to the x,y,z, and w axes, while the bottom plot shows angular velocity in x,y, and z directions.

In order to utilize ROS across multiple machines, each machine must be networked in some way. For this example, the Powerbot was connected to a laptop across an ad-hoc network named “ANTros2,” which is reflected in Figure IV.6. The results are precisely the same as for the single host case only with two hosts in this scenario. The script is running

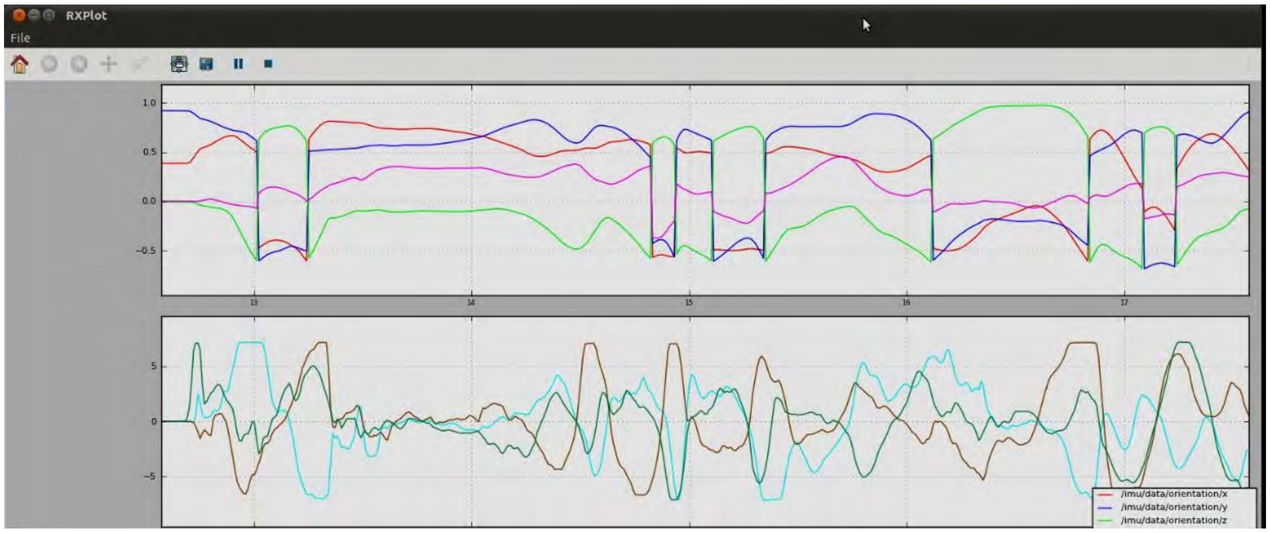
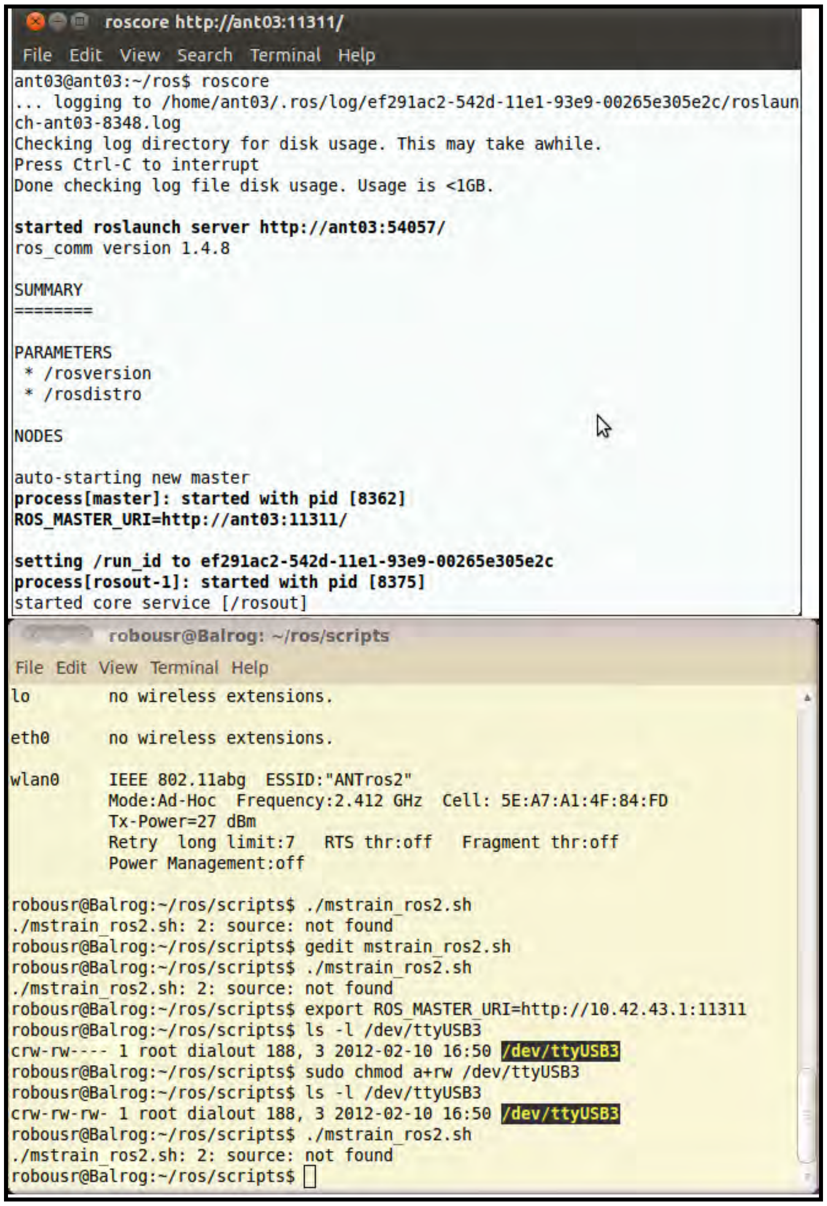


Figure IV.5: Plot of AHRS Orientation (top) and Angular Velocity (bottom)

on the Powerbot and *roscore* is being executed on the laptop, which is acting as the base station.

4.1.2 udev. Using the information from the syslog file and hwinfo previously used, udevadm was used to gather field data to use in the udev rules for each sensor. Any combination of these fields can be used to produce the rule. For the AHRS, the "idVendor", "idProduct", and "serial" fields were used. A simple initializing script was written to wrap the ROS AHRS script previously introduced. This script was set to execute whenever a device meeting the stated criteria was mounted to and location beginning with /dev/ttyUSB. For example, this would work if the serial converter were mounted to /dev/ttyUSB3, /dev/ttyUSB0, etc. because of the wild card ('?' character) that was used in the rule. The following, Figure IV.8 shows the udev rule in its entirety.



roscore http://ant03:11311/

File Edit View Search Terminal Help

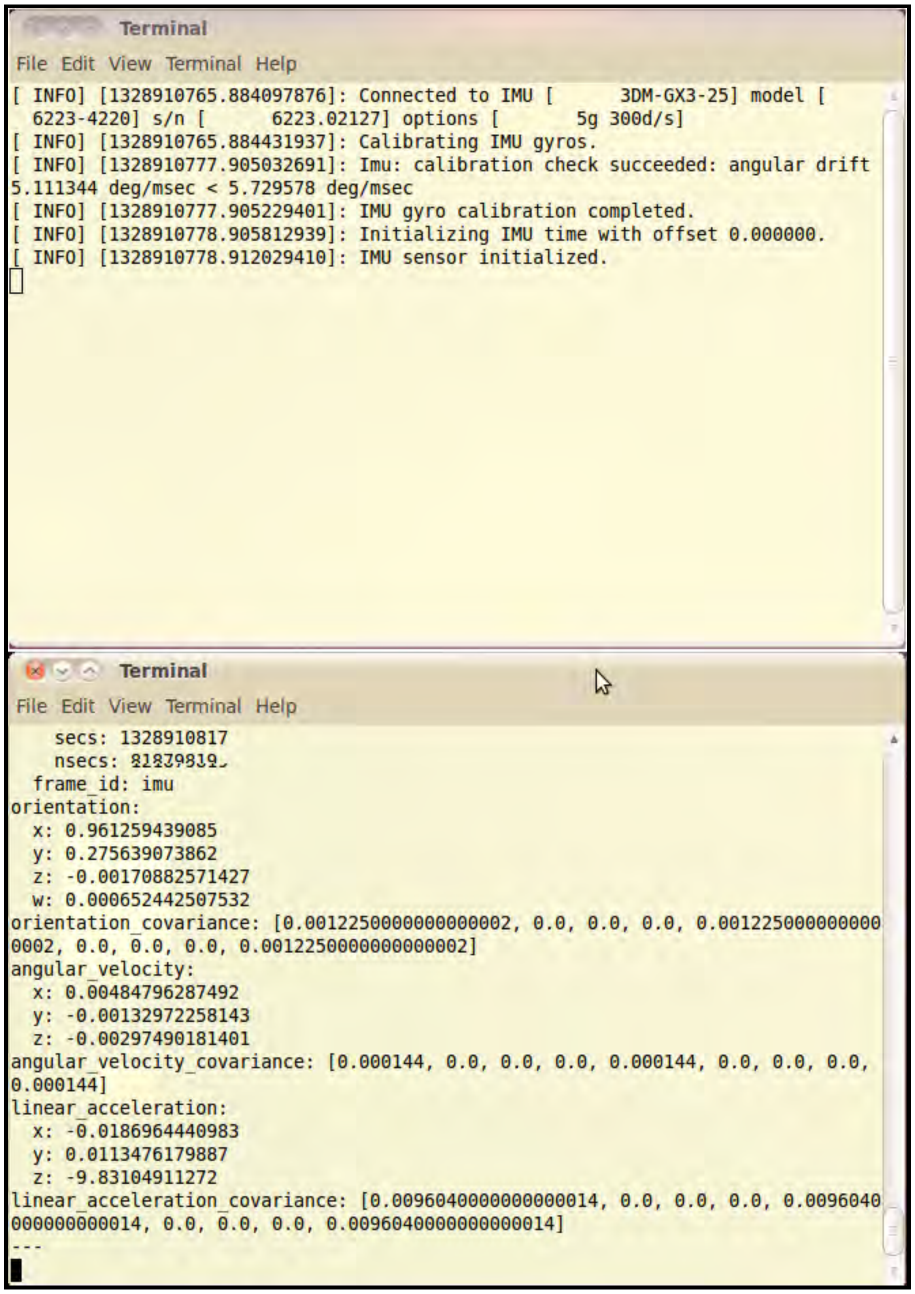
```
ant03@ant03:~/ros$ roscore
... logging to /home/ant03/.ros/log/ef291ac2-542d-11e1-93e9-00265e305e2c/roslaun
ch-ant03-8348.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ant03:54057/
ros_comm version 1.4.8

SUMMARY
========

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [8362]
ROS_MASTER_URI=http://ant03:11311/

setting /run_id to ef291ac2-542d-11e1-93e9-00265e305e2c
process[rosout-1]: started with pid [8375]
started core service [/rosout]
```

robousr@Balrog: ~/ros/scripts

File Edit View Terminal Help

```
lo        no wireless extensions.

eth0      no wireless extensions.

wlan0     IEEE 802.11abg  ESSID:"ANTros2"
          Mode:Ad-Hoc  Frequency:2.412 GHz  Cell: 5E:A7:A1:4F:84:FD
          Tx-Power=27 dBm
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:off

robousr@Balrog:~/ros/scripts$ ./mstrain_ros2.sh
./mstrain_ros2.sh: 2: source: not found
robousr@Balrog:~/ros/scripts$ gedit mstrain_ros2.sh
robousr@Balrog:~/ros/scripts$ ./mstrain_ros2.sh
./mstrain_ros2.sh: 2: source: not found
robousr@Balrog:~/ros/scripts$ export ROS_MASTER_URI=http://10.42.43.1:11311
robousr@Balrog:~/ros/scripts$ ls -l /dev/ttyUSB3
crw-rw---- 1 root dialout 188, 3 2012-02-10 16:50 /dev/ttyUSB3
robousr@Balrog:~/ros/scripts$ sudo chmod a+rw /dev/ttyUSB3
robousr@Balrog:~/ros/scripts$ ls -l /dev/ttyUSB3
crw-rw-rw- 1 root dialout 188, 3 2012-02-10 16:50 /dev/ttyUSB3
robousr@Balrog:~/ros/scripts$ ./mstrain_ros2.sh
./mstrain_ros2.sh: 2: source: not found
robousr@Balrog:~/ros/scripts$ 
```

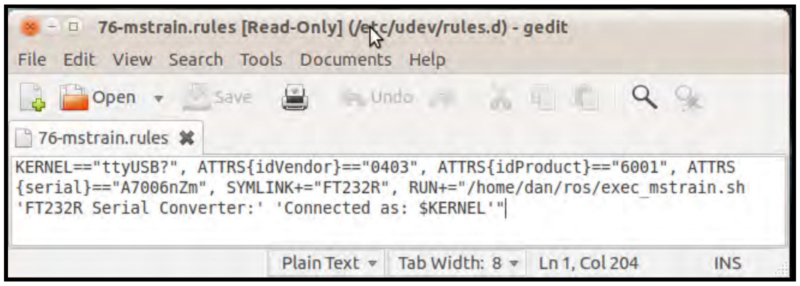
Figure IV.6: Execution of AHRS script with *roscore* running on separate machine



```
[ INFO] [1328910765.884097876]: Connected to IMU [      3DM-GX3-25] model [
  6223-4220] s/n [      6223.02127] options [       5g 300d/s]
[ INFO] [1328910765.884431937]: Calibrating IMU gyros.
[ INFO] [1328910777.905032691]: Imu: calibration check succeeded: angular drift
5.111344 deg/msec < 5.729578 deg/msec
[ INFO] [1328910777.905229401]: IMU gyro calibration completed.
[ INFO] [1328910778.905812939]: Initializing IMU time with offset 0.000000.
[ INFO] [1328910778.912029410]: IMU sensor initialized.
```

```
    secs: 1328910817
    nsecs: 81879819
  frame_id: imu
orientation:
  x: 0.961259439085
  y: 0.275639073862
  z: -0.00170882571427
  w: 0.000652442507532
orientation_covariance: [0.0012250000000000002, 0.0, 0.0, 0.0, 0.001225000000000
0002, 0.0, 0.0, 0.0, 0.0012250000000000002]
angular_velocity:
  x: 0.00484796287492
  y: -0.00132972258143
  z: -0.00297490181401
angular_velocity_covariance: [0.000144, 0.0, 0.0, 0.0, 0.000144, 0.0, 0.0, 0.0,
0.000144]
linear_acceleration:
  x: -0.0186964440983
  y: 0.0113476179887
  z: -9.83104911272
linear_acceleration_covariance: [0.0096040000000000014, 0.0, 0.0, 0.0, 0.0096040
000000000014, 0.0, 0.0, 0.0, 0.0096040000000000014]
---
```

Figure IV.7: Output observed using AHRS script with *roscore* running on separate machine


76-mstrain.rules [Read-Only] (/etc/udev/rules.d) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
76-mstrain.rules
Plain Text
Tab Width: 8
Ln 1, Col 204
INS

```
KERNEL=="ttyUSB?", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", ATTRS
{serial}=="A7006nZm", SYMLINK+="FT232R", RUN+="/home/dan/ros/exec_mstrain.sh
'FT232R Serial Converter:' 'Connected as: $KERNEL'"
```

Figure IV.8: udev rule for MicroStrain AHRS

The udev rule fails to execute the script as expected. Some troubleshooting was performed in order to try and understand why this is happening. The udev rule was first attempted prior to the scripts being set as executables, which led to an error. Obviously, the script cannot be executed if it isn't set as an executable in the first place. The system log file shows the error, which demonstrates the rule attempted to execute the script. The relevant portion of the file follows:

```
[  104.560364] usb 1-6.4.1.4: new full speed USB device using ehci_hcd and address 20
[  104.691182] ftdi_sio 1-6.4.1.4:1.0: FTDI USB Serial Device converter detected
[  104.691264] usb 1-6.4.1.4: Detected FT232RL
[  104.691270] usb 1-6.4.1.4: Number of endpoints 2
[  104.691276] usb 1-6.4.1.4: Endpoint 1 MaxPacketSize 64
[  104.691281] usb 1-6.4.1.4: Endpoint 2 MaxPacketSize 64
[  104.691285] usb 1-6.4.1.4: Setting MaxPacketSize 64
[  104.691684] usb 1-6.4.1.4: FTDI USB Serial Device converter now attached to ttyUSB3
[2391]: exec of program '/home/dan/ros/exec_mstrain.sh' failed
[  145.342600] #
```

After changing the file as an executable, the introduction of the AHRS via converter was attempted again, this time yielding no error. However, getting rid of the error did not cause the script to execute. The fact that no error was produced indicates the rule was successful in that the script was attempted to be run and it was accessible. However, there must be some sort of problem in the implementation, possibly in the linking of scripts, that prevents the scripts from executing. This has yet to be resolved, but is left as a recommendation for future work.

4.1.3 Assumptions. It is quite obvious that in order to do any of this in the first place, some assumptions must be made. For starters, the converter device connected to the given location is assumed to be used with the AHRS and that device only even though it is a 4-port converter device. Secondly, it is assumed the device will be mounted to /dev/ttyUSB3 for the ROS script since there is no way to utilize wild cards for this. It is a good assumption based on trial-and-error, observing where the host OS usually mounts the device. Finally, all of the tests operate under the awareness of what the actual device is that is being mounted. Without this information, the script could not be written and the ROS packages for each device could not be pre-installed on the host OS prior to using them.

4.2 Data Handling

The proposed message passing scheme and overall structure of an adaptable PnP system uses a Publish-Subscribe paradigm. When viewing the software artifacts through the lens of design patterns, this represents the Mediator pattern as there are one-to-many subscribers and multiple publishers [11]. There are several ways to achieve this result and these will be explored in this section. Results will be presented in the order of preference. Not surprisingly, the discussion will begin with what is available using ROS as its native mode of communication is precisely that which is desired. This will be followed by what is currently available and more traditional approaches that could be taken to achieve similar results. All of these will be compared, highlighting the tradeoffs that must be made when selecting one approach over another.

4.2.1 ROS. ROS natively operates under a publish / subscribe message passing paradigm. For example, going back the the results of the AHRS script, data is not only output to the screen it is also being published to a topic. These “topics” are in essence the “moderators” according to the pattern. They serve as the platform for communication amongst all entities running within ROS. These entities, known as “nodes” in ROS, have a very narrow scope of control meaning they are very specialized in nature [49]. In the software engineering world this is indicates low coupling, high cohesion. This is precisely what is desired as there are many unique sensors in a proposed PnP system chosen to provide unique and distinct contributions. Wikipedia provides a very good description of coupling and in their description the manner in which ROS is postured reflects that of “data coupling” [51]. This is captured in Figure IV.9. The object, or data type, used to pass information in ROS are known as messages and they are published and subscribed by nodes across topics. These concepts are so basic and fundamental to ROS that even */rosout*, which is the standard output in ROS, is set up in the same fashion with *rosout* as the node and */rosout* as the topic [50].

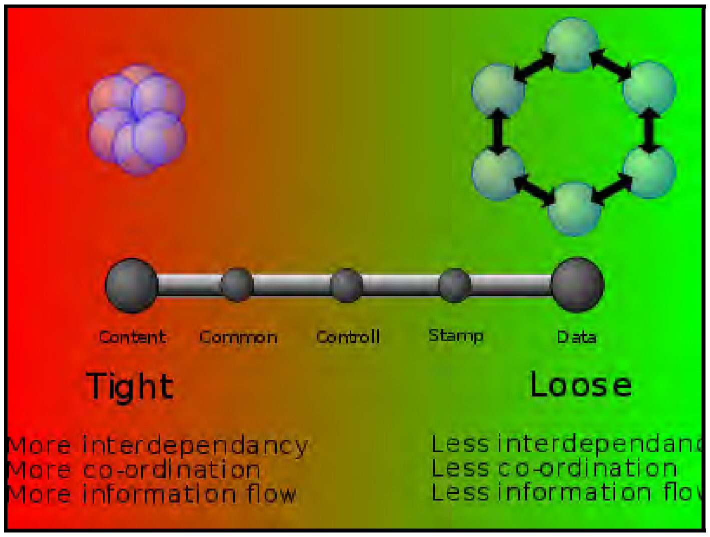
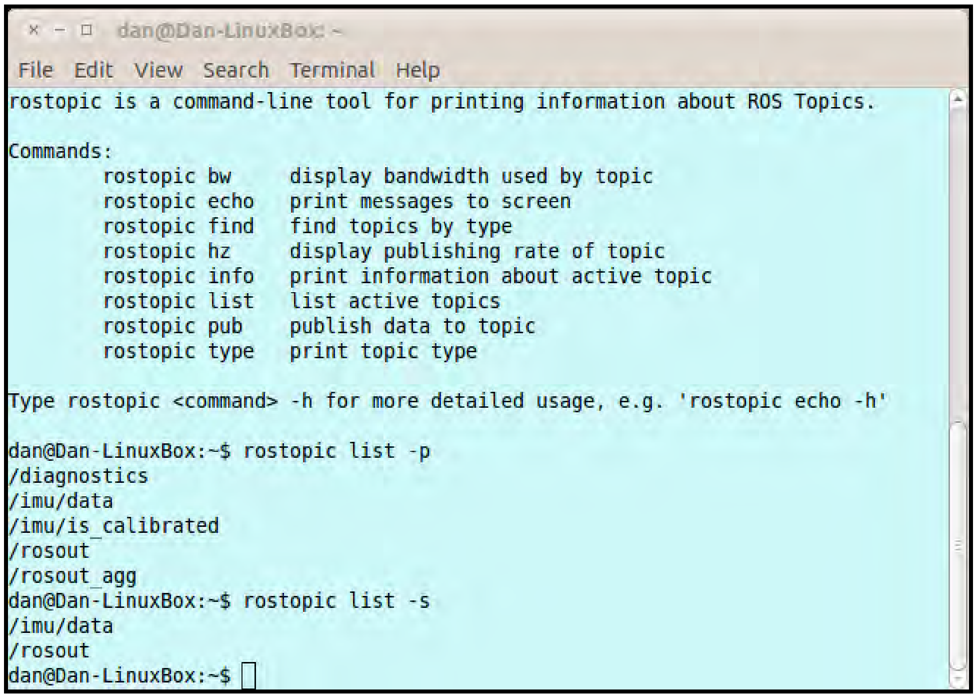


Figure IV.9: Software Coupling [51]

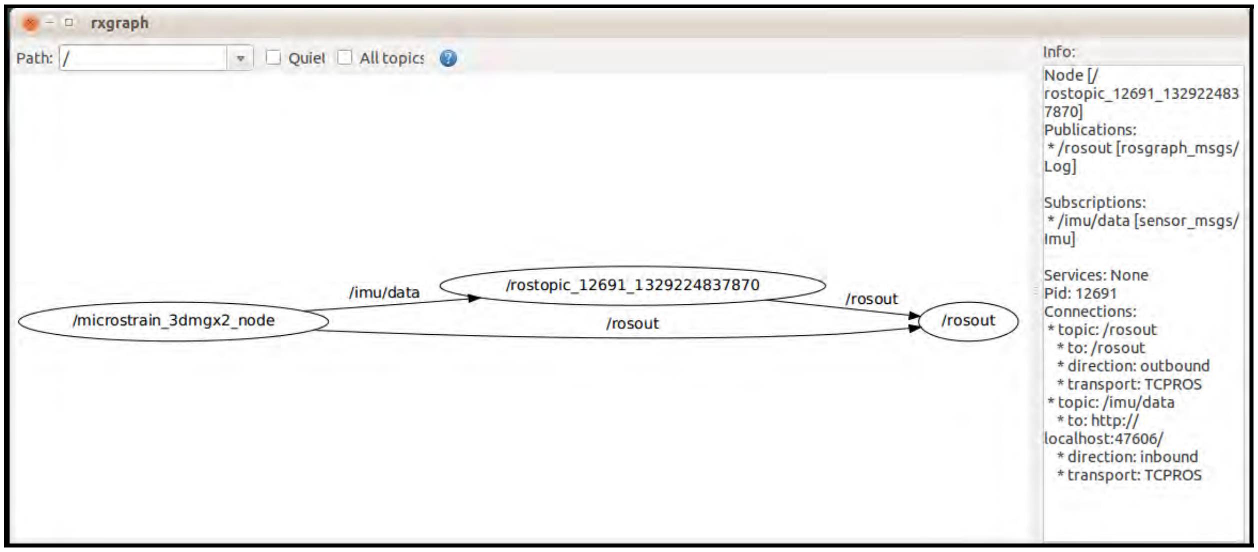
The fact that ROS is organized in this manner allows it to be used directly in a PnP sensor application. In the previous examples using the AHRS, the data is published to the imu/data topic using the imu node. The *rosout* node is subscribing to the imu/data topic and displaying this in real time. The following figures help demonstrate this using ROS commands to list the nodes and topics existing while the program is running. Figure IV.10 is a screenshot showing the available *rostopic* commands, along with a listing of all active publishers and subscribers. Figure IV.11 is a graphical representation of this with the nodes displayed as circles and the topics as lines.



```
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
        rostopic bw     display bandwidth used by topic
        rostopic echo   print messages to screen
        rostopic find   find topics by type
        rostopic hz     display publishing rate of topic
        rostopic info   print information about active topic
        rostopic list   list active topics
        rostopic pub    publish data to topic
        rostopic type   print topic type

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'

dan@Dan-LinuxBox:~$ rostopic list -p
/diagnostics
/imu/data
/imu/is_calibrated
/rosout
/rosout_agg
dan@Dan-LinuxBox:~$ rostopic list -s
/imu/data
/rosout
dan@Dan-LinuxBox:~$
```

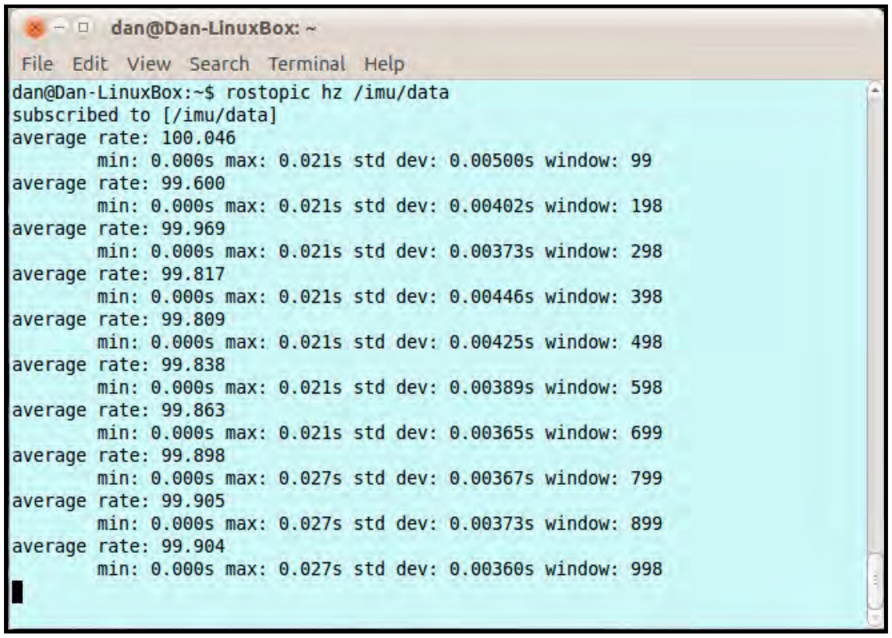
Figure IV.10: *rostopic* commands and listing of publishers and subscribers

Another interesting tool ros has in terms of analyzing topics is the *rostopic hz* command. This displays information about the “real time” aspect of the messages being generated. For example, using this command for the imu/data topic shows the rate in which the data is being published. In doing so, the diagnostic tool itself is subscribing to the topic. The results of this are shown in Figure IV.12. Figure IV.13 displays an updated graph of all of the nodes, reflecting the addition of the new subscriber.



```
Info:
Node [/
rostopic_12691_132922483
7870]
Publications:
 * /rosout [rosgraph_msgs/
Log]

Subscriptions:
 * /imu/data [sensor_msgs/
Imu]

Services: None
Pid: 12691
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
 * topic: /imu/data
   * to: http://
localhost:47606/
   * direction: inbound
   * transport: TCPROS
```

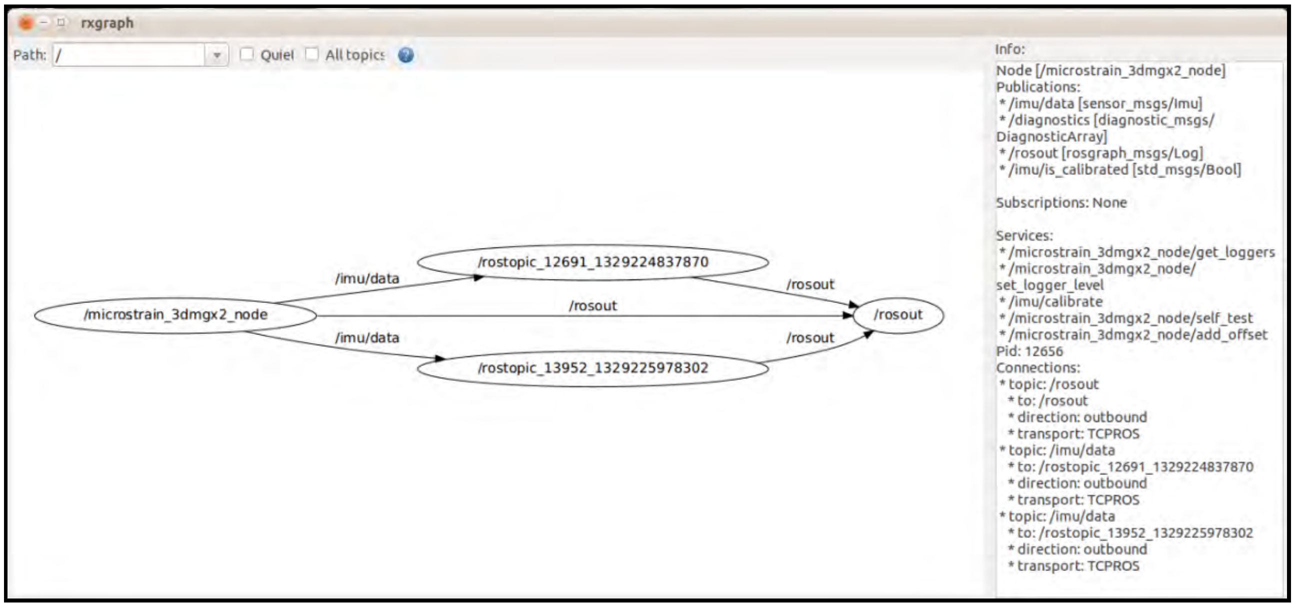
Figure IV.11: Graph of publishers, subscribers, and topics when AHRS software is running



```
dan@Dan-LinuxBox:~$ rostopic hz /imu/data
subscribed to [/imu/data]
average rate: 100.046
        min: 0.000s max: 0.021s std dev: 0.00500s window: 99
average rate: 99.600
        min: 0.000s max: 0.021s std dev: 0.00402s window: 198
average rate: 99.969
        min: 0.000s max: 0.021s std dev: 0.00373s window: 298
average rate: 99.817
        min: 0.000s max: 0.021s std dev: 0.00446s window: 398
average rate: 99.809
        min: 0.000s max: 0.021s std dev: 0.00425s window: 498
average rate: 99.838
        min: 0.000s max: 0.021s std dev: 0.00389s window: 598
average rate: 99.863
        min: 0.000s max: 0.021s std dev: 0.00365s window: 699
average rate: 99.898
        min: 0.000s max: 0.027s std dev: 0.00367s window: 799
average rate: 99.905
        min: 0.000s max: 0.027s std dev: 0.00373s window: 899
average rate: 99.904
        min: 0.000s max: 0.027s std dev: 0.00360s window: 998
```

Figure IV.12: Results of running *rostopic hz* on the imu/data topic

All of these examples are nice, but they don't really solve any real navigation problems. That fact is not a coincidence, however, as the whole point of the problem was to provide an architecture that is scalable, dynamic and not tied to any one specific



```
Info:
Node [/microstrain_3dmgx2_node]
Publications:
 * /imu/data [sensor_msgs/Imu]
 * /diagnostics [diagnostic_msgs/
DiagnosticArray]
 * /rosout [rosgraph_msgs/Log]
 * /imu/is_calibrated [std_msgs/Bool]

Subscriptions: None

Services:
 * /microstrain_3dmgx2_node/get_loggers
 * /microstrain_3dmgx2_node/
set_logger_level
 * /imu/calibrate
 * /microstrain_3dmgx2_node/self_test
 * /microstrain_3dmgx2_node/add_offset
Pid: 12656
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
 * topic: /imu/data
   * to: /rostopic_12691_1329224837870
   * direction: outbound
   * transport: TCPROS
 * topic: /imu/data
   * to: /rostopic_13952_1329225978302
   * direction: outbound
   * transport: TCPROS
```

Figure IV.13: Updated graph reflecting the addition of a new node / subscriber

application. Basically, the use of ROS along with OS tools capable of addressing autonomous behavior achieves every metric desired in the original PnP problem statement. That is, sensors are discovered and configured when they are plugged into the system, drivers are executed so they will function, and data is published and handled by an Observer. Furthermore, all of the publishers are tracked in real time and any application software can request an up-to-date listing of all available sensors and subscribe to whatever ones are needed. There aren't really any limitations on the numbers of subscribers either for all practical purposes. Assuming all three of the test sensors are connected to the PnP system (camera, AHRS, and LADAR), all of them will be set to publish to topics. Application software, such as a Kalman Filtering program, can then subscribe to these topics directly through ROS. Of course this requires the use of client-specific APIs and ROS being installed and configured on any machine executing ROS commands. There are two major client APIs in ROS: *roscpp* and *rospy*. As the names suggest, these are APIs for the integration of C++ and Python, respectively. It should be noted there is also rosjava, but it isn't as widely used as the other two. All of the work and experimentation performed as part of this thesis was accomplished using *roscpp*.

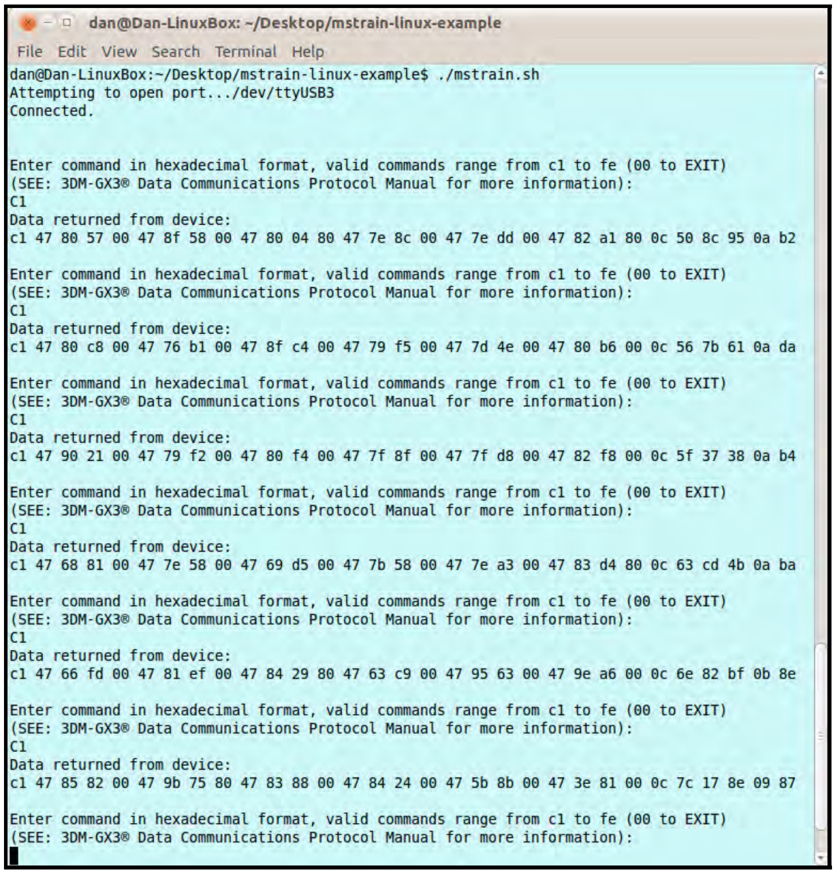
With a PnP system capable of providing data in a flexible and responsive manner utilizing the publish / subscribe message-passing scheme, the writer of the application software, such as an implementation of a positioning algorithm, can use these sensors by subscribing them using one of the APIs. Obviously, if a program was written in C++, *roscpp* should be used; similarly *rosjava* should be used for Java programs, etc. All of these are intuitive, user-friendly, and have well-documented instructions for use providing the writer is familiar with ROS. The proposed ROS PnP system is thus design-time static and run-time dynamic [10]. Simply put, all of the information regarding any possible sensors to be used in an application must be known at design time so that the drivers and software for running the device will be available should it be added to the system. In this sense, the design is static. However, within the realm of all possible sensors that were decided at design time, any number of them can be used at run-time. This isn't limited to sensors that are already connected to a system as adding a new sensor that was accounted for a at design time can be used without interrupting a particular test or application.

4.2.2 Traditional Approaches. There are two traditional approaches to look at: a pure component-based implementation and that which is already in use at the ANT center. Since this is all experimental, this subsection will start by observing a component-driven design approach and then move into the system that is currently employed. The reason a component-driven approach was chosen as part of this research is, not only due to the fact that an OO approach is part of the system currently being employed, but also due to the promise it foretells when reflecting on the Fortier paper discussed in Chapter 3. The overall functionality needs do not change of course, only the implementation and the advantages provided in the chosen approach. At a minimum, the system must recognize when a device is connected or removed, it must know something about that device, and it must use software to make it work. Finally, the data that is retrieved by the sensor must be accessible in some way by the application software using it.

In a component-based system, a script is still the best approach for automating the initial device recognition and configuration. The code for the script would not be much different than that for ROS, although the ROS commands would not be required obviously. In this test case, the execution command is really all that's required, since we're working under the assumptions listed previously. The driver used for this part is the Linux driver provided by the device manufacturer on the product page. The code was compiled and the executable is named "mstrain." The executable requires an argument specifying the sensor location, which is /dev/ttyUSB3 in this case. The results are a bit different than those observed with ROS as the driver is much more of a bare-bones type. This is being run interactively for demonstration purposes only. Obviously, if this approach were to be used in generating a PnP solution, a separate component, such as the drive handler mentioned in Chapter 3, would interface with the driver directly. Figure IV.14 shows the results of this part of the test run.

The input and output are also a bit different than what was observed with the ROS driver — even a bit cryptic. The 3DM-GX3-25 Data Communications Manual must be followed in order to utilize the sensor in the desired way and to understand the data being returned. All of the commands are entered in hexadecimal and the results are output in hex as well. The mode chosen for this example is "raw data" which returns only one data set at a time. A command of 0xC1 specifies this; it can be seen where this is manually entered in Figure IV.14 and the corresponding result. It should be noted that a continuous mode is possible as well and can be set using a different command. Table IV.1 provides an explanation of the data returned after giving the 0xC1 command.

Obviously there should be a separate program that handles this data and packages it up into a useful form. It can then be passed on to a an Observer entity that will be

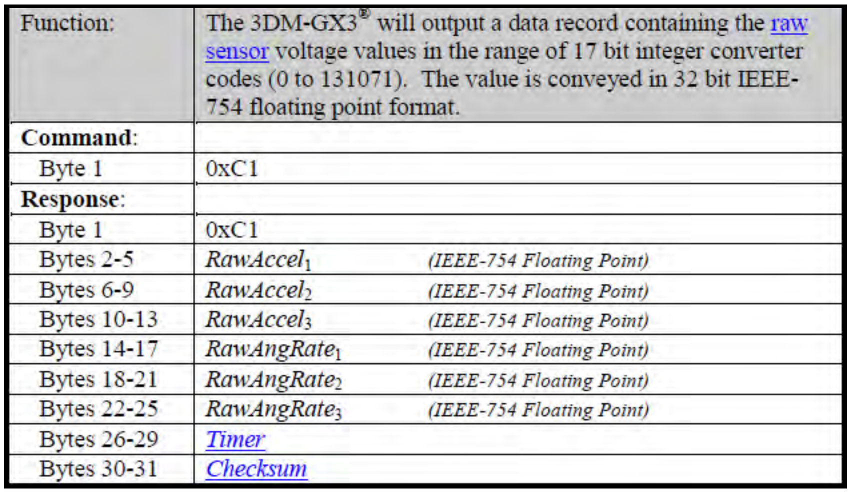


```
dan@Dan-LinuxBox:~/Desktop/mstrain-linux-example$ ./mstrain.sh
Attempting to open port.../dev/ttyUSB3
Connected.


Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 80 57 00 47 8f 58 00 47 80 04 80 47 7e 8c 00 47 7e dd 00 47 82 a1 80 0c 50 8c 95 0a b2

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 80 c8 00 47 76 b1 00 47 8f c4 00 47 79 f5 00 47 7d 4e 00 47 80 b6 00 0c 56 7b 61 0a da

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 90 21 00 47 79 f2 00 47 80 f4 00 47 7f 8f 00 47 7f d8 00 47 82 f8 00 0c 5f 37 38 0a b4

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 68 81 00 47 7e 58 00 47 69 d5 00 47 7b 58 00 47 7e a3 00 47 83 d4 80 0c 63 cd 4b 0a ba

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 66 fd 00 47 81 ef 00 47 84 29 80 47 63 c9 00 47 95 63 00 47 9e a6 00 0c 6e 82 bf 0b 8e

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
C1
Data returned from device:
c1 47 85 82 00 47 9b 75 80 47 83 88 00 47 84 24 00 47 5b 8b 00 47 3e 81 00 0c 7c 17 8e 09 87

Enter command in hexadecimal format, valid commands range from c1 to fe (00 to EXIT)
(SEE: 3DM-GX3® Data Communications Protocol Manual for more information):
█
```

dan@Dan-LinuxBox: ~/Desktop/mstrain-linux-example
File Edit View Search Terminal Help

Figure IV.14: Results of running the MicroStrain AHRS using bare-bones driver

Table IV.1: Raw Accelerometer and Angular Rate Sensor Outputs [25]



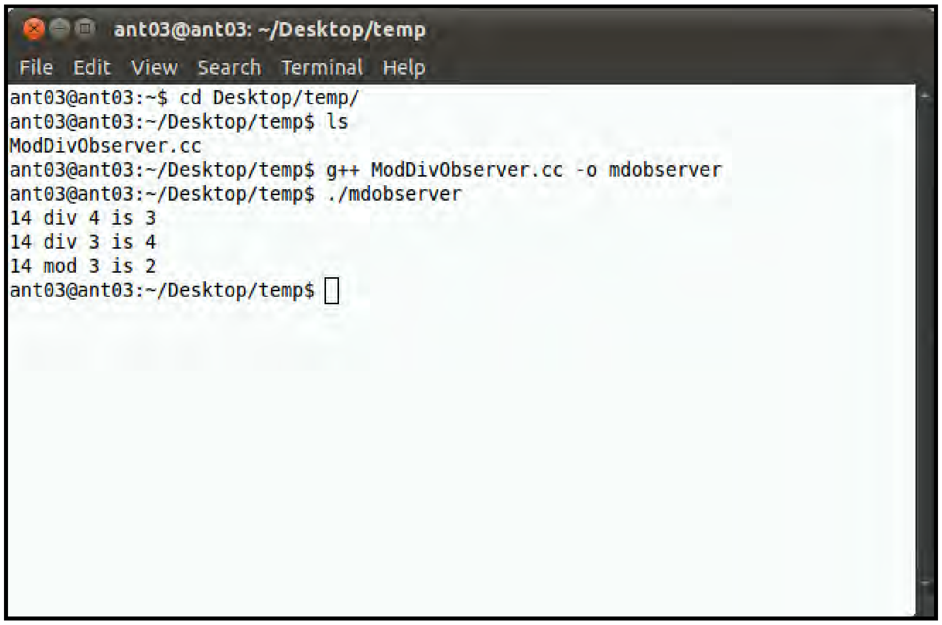
Function:	The 3DM-GX3® will output a data record containing the raw sensor voltage values in the range of 17 bit integer converter codes (0 to 131071). The value is conveyed in 32 bit IEEE-754 floating point format.	
Command:		
Byte 1	0xC1	
Response:		
Byte 1	0xC1	
Bytes 2-5	*RawAccel*$_1$	*(IEEE-754 Floating Point)*
Bytes 6-9	*RawAccel*$_2$	*(IEEE-754 Floating Point)*
Bytes 10-13	*RawAccel*$_3$	*(IEEE-754 Floating Point)*
Bytes 14-17	*RawAngRate*$_1$	*(IEEE-754 Floating Point)*
Bytes 18-21	*RawAngRate*$_2$	*(IEEE-754 Floating Point)*
Bytes 22-25	*RawAngRate*$_3$	*(IEEE-754 Floating Point)*
Bytes 26-29	*Timer*	
Bytes 30-31	*Checksum*	

responsible for handling publish / subscribe requests. For the purpose of this experiment and in the interest of time and brevity, this part of the software will be skipped and it will be assumed there is a data structure storing these sensor values as they come in at a specific interval. The attention then turns to the Observer and the manner in which it will handle requests.

Some examples that are freely available on the internet were used to attempt communication between software artifacts in order to observe the complexities and benefits of using the design pattern in this application. These results provide only examples of how an Observer can be implemented and how well it performed. The design and implementation of an Observer pattern within a completed software suite for the PnP application was attempted, however, this falls outside the scope of this effort and is left for possible future work. In any event, some of the key results follow and will be described in terms of their relation to requirements of a PnP system. To start, a simple Subject/Observer example follows which shows how an observer was implemented in C++ for calculating divisor (div) and modulo (mod). The code for the example was taken from sourcemaking.com and can be found in Appendix: Code, Section 1.1. [40]. The result of running this program is shown in Figure IV.15.

This Observer example does not represent a publish / subscribe message passing scheme but it does help to solidify what an Observer actually does. The subject notifies the observers of value changes and the Observers. The Observers can do whatever with the data at that point but in this example it is being used in simple calculations, the results of which the Observers output to the screen as shown in Figure IV.15. Other examples were attempted as well with similar results. They are not listed here because these really don't represent the publish / subscribe message-passing scheme desired either.

The ability to create the environment desired in C++ using an Observer pattern is not trivial. There is a lot of scaffolding required in order to even present working code that


ant03@ant03: ~/Desktop/temp
File Edit View Search Terminal Help

```
ant03@ant03:~$ cd Desktop/temp/
ant03@ant03:~/Desktop/temp$ ls
ModDivObserver.cc
ant03@ant03:~/Desktop/temp$ g++ ModDivObserver.cc -o mdobserver
ant03@ant03:~/Desktop/temp$ ./mdobserver
14 div 4 is 3
14 div 3 is 4
14 mod 3 is 2
ant03@ant03:~/Desktop/temp$ 
```

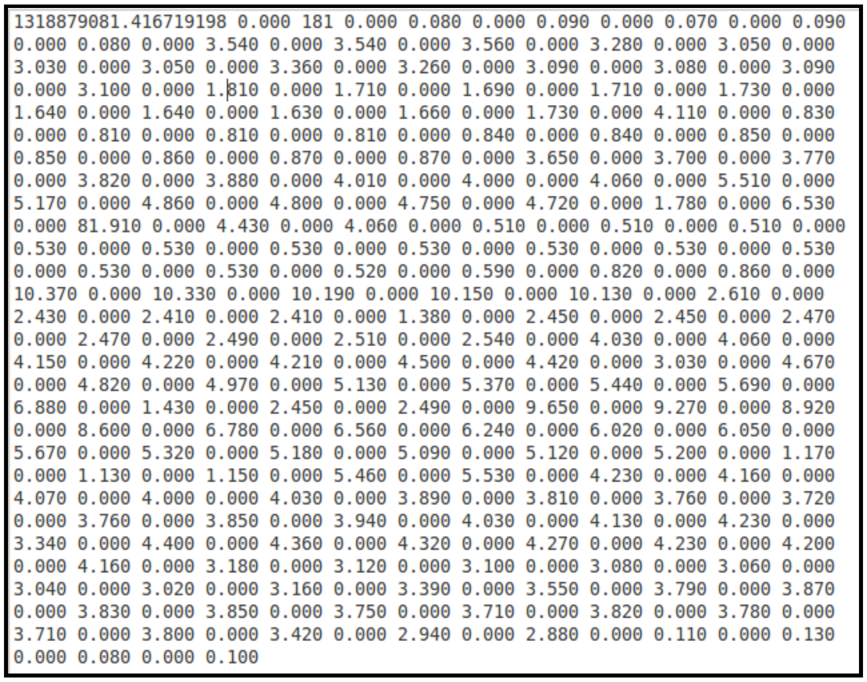
Figure IV.15: Results of running mathematical Observer example

was extensible and scalable. The idea is to keep the implementation details separate from the structure and capabilities given by the Observer so the use of templates was a primary necessity. This is a lot of work to implement but doesn't seem to add much, short of academically, since ROS has already been developed and works quite well. Python or Java may be a better option than C++ if this were to be implemented in future work. Python is an obvious choice as it is very user-friendly and Java has its own Observer interface.

4.3 Current ANT Software - NextGen VAN

The design pattern methodology set forth by the Gang of Four is very noticeable in the NextGen VAN software currently employed in AFIT's ANT center. It is quite functional and was able to be used out-of-the-box for the tests conducted. The structure of the artifacts is quite cumbersome, however, and much time was spent trying to figure out what needed to be done in order to tailor a specific setup for a particular test run. Furthermore, once the decision has been made as to what sensors are being used for a particular test run, it is completely fixed and static at that point. In other words, it is design time dynamic in that any new sensor can be added by using the right hooks in the main program and loading the proper configuration files. However, the solution is run-time static since the sensors are hard-coded to the implementation at compile-time.

For the test, only two features were utilized: teleop and laser scanning. To specify this, at design time (or prior to compiling the program) all of the other hooks were commented out and the main program was compiled with the teleop and SICK LADAR features selected. The PowerBot was driven around using the keyboard with no problems, however, the test did not go beyond the physical limitations of the USB keyboard. The SICK laser actively collected data and the results were saved to a local file. A sample of the output is given in Figure IV.16.



```
1318879081.416719198 0.000 181 0.000 0.080 0.000 0.090 0.000 0.070 0.000 0.090
0.000 0.080 0.000 3.540 0.000 3.540 0.000 3.560 0.000 3.280 0.000 3.050 0.000
3.030 0.000 3.050 0.000 3.360 0.000 3.260 0.000 3.090 0.000 3.080 0.000 3.090
0.000 3.100 0.000 1.810 0.000 1.710 0.000 1.690 0.000 1.710 0.000 1.730 0.000
1.640 0.000 1.640 0.000 1.630 0.000 1.660 0.000 1.730 0.000 4.110 0.000 0.830
0.000 0.810 0.000 0.810 0.000 0.810 0.000 0.840 0.000 0.840 0.000 0.850 0.000
0.850 0.000 0.860 0.000 0.870 0.000 0.870 0.000 3.650 0.000 3.700 0.000 3.770
0.000 3.820 0.000 3.880 0.000 4.010 0.000 4.000 0.000 4.060 0.000 5.510 0.000
5.170 0.000 4.860 0.000 4.800 0.000 4.750 0.000 4.720 0.000 1.780 0.000 6.530
0.000 81.910 0.000 4.430 0.000 4.060 0.000 0.510 0.000 0.510 0.000 0.510 0.000
0.530 0.000 0.530 0.000 0.530 0.000 0.530 0.000 0.530 0.000 0.530 0.000 0.530
0.000 0.530 0.000 0.530 0.000 0.520 0.000 0.590 0.000 0.820 0.000 0.860 0.000
10.370 0.000 10.330 0.000 10.190 0.000 10.150 0.000 10.130 0.000 2.610 0.000
2.430 0.000 2.410 0.000 2.410 0.000 1.380 0.000 2.450 0.000 2.450 0.000 2.470
0.000 2.470 0.000 2.490 0.000 2.510 0.000 2.540 0.000 4.030 0.000 4.060 0.000
4.150 0.000 4.220 0.000 4.210 0.000 4.500 0.000 4.420 0.000 3.030 0.000 4.670
0.000 4.820 0.000 4.970 0.000 5.130 0.000 5.370 0.000 5.440 0.000 5.690 0.000
6.880 0.000 1.430 0.000 2.450 0.000 2.490 0.000 9.650 0.000 9.270 0.000 8.920
0.000 8.600 0.000 6.780 0.000 6.560 0.000 6.240 0.000 6.020 0.000 6.050 0.000
5.670 0.000 5.320 0.000 5.180 0.000 5.090 0.000 5.120 0.000 5.200 0.000 1.170
0.000 1.130 0.000 1.150 0.000 5.460 0.000 5.530 0.000 4.230 0.000 4.160 0.000
4.070 0.000 4.000 0.000 4.030 0.000 3.890 0.000 3.810 0.000 3.760 0.000 3.720
0.000 3.760 0.000 3.850 0.000 3.940 0.000 4.030 0.000 4.130 0.000 4.230 0.000
3.340 0.000 4.400 0.000 4.360 0.000 4.320 0.000 4.270 0.000 4.230 0.000 4.200
0.000 4.160 0.000 3.180 0.000 3.120 0.000 3.100 0.000 3.080 0.000 3.060 0.000
3.040 0.000 3.020 0.000 3.160 0.000 3.390 0.000 3.550 0.000 3.790 0.000 3.870
0.000 3.830 0.000 3.850 0.000 3.750 0.000 3.710 0.000 3.820 0.000 3.780 0.000
3.710 0.000 3.800 0.000 3.420 0.000 2.940 0.000 2.880 0.000 0.110 0.000 0.130
0.000 0.080 0.000 0.100
```

Figure IV.16: Results of running SICK laser on PowerBot using NextGen VAN

It should be noted there were many more lines than this in the text file for the laser data. The test ran for approximately 2 minutes and there were well over 200 lines of data in the format shown. Also, this is purely raw data which needs to be processed in some way to be usable, which is the responsibility of the using applications. However, time stamps are missing, which would help with real-time utilization of the code for a positioning algorithm or even post-processing or data analysis. Adding a few lines of code to the “Sick_lms_2xx_lidar.cpp” where the data is being collected could allow for this.

As an example, using *time* or *ctime* in C++ could be used to get the current time in seconds at the beginning of each data stream. This really isn’t much good for sensor measurements, such as the SICK laser, since they are taken much faster than once per second. A higher resolution is therefore needed to distinguish one measurement from another, preventing duplicate time stamps for different data sets. In Linux, one system feature that can help provide this is the *gettimeofday()* function, which allows for up to microsecond resolution. This is a unix-specific function available through the use of the *sys/time.h* header file [28]. An example of the code required to do this follows.

```
time_t seconds;

seconds = time (NULL);

struct timeval detail_time;
gettimeofday(&detail_time ,NULL);
cout << seconds << "." << detail_time.tv_usec/1000 << endl;   //milliseconds
cout << seconds << "." << detail_time.tv_usec << endl; //microseconds
```

Currently, this just prints the time stamps to the screen. If it were used in tandem with the data collection from the SICK, the time stamps need to be output wherever the data is. This can be done in a similar way as shown by outputting to a text file as opposed to stdout, however, it is probably more useful to create a new field in the data structure being used to hold the SICK data and adding the time stamps directly to that.

One other part of the NextGen VAN software that cannot fully be captured in a textual document is just how large and unwieldy the structure of the software itself actually is. To somewhat put things into perspective, there are 262 total directories and 1,377 total files within the main folder of the software used. Granted, only a fraction of these would a person need to know about explicitly in order to use the software effectively, however, it does make navigating the file system quite difficult and timely. For a system using ROS as middleware, much of the complexity is abstracted out. What's more is that ROS commands are fully integrated with the command line and there are many tools which are helpful in finding and instantiating packages.

As a further comparison in terms of time, the ROS-based system was able to configure and gather data from the Prosilica Camera and Microstrain AHRS, as well as manage that data according to a publish / subscribe message-passing scheme, in just over one minute. For NextGen VAN, setting up the software to save the SICK data to a file and allow the PowerBot to be driven with a keyboard took nearly 10 minutes, and that was with one of the people most familiar with the software performing all of the setup.

4.4 ROS Usability Limitations

ROS was already introduced quite extensively in Section 4.2 on page 83, as well as various other portions of the document. However, simply relaying the results of a particular successful test run does not paint an accurate picture of usability of a particular approach — in this case ROS. This section seeks to provide that missing information, describing some of the results of trying to use the environment and, in particular, some of the problems encountered along the way.

4.5 Platform Limitations

ROS has OS limitations that created somewhat of a problem from the onset. Initially, the hardware testing platform was a Pioneer 2AT that has been utilized in many different navigation tests in the past. It is much smaller and cheaper to maintain than the PowerBot which made it an appealing choice to conduct this research. However, the hardware on the robot wasn't current enough to run Ubuntu 10.04+. The most recent version that was successfully installed is 9.10 - Karmic Koala. Essentially, this alone prevented ROS from being used since the two most-recent and supported ROS builds, code-named diamondback and electric, were developed for Ubuntu 10.04+ (through 11.10 at the time of this writing). Pre-compiled binaries (.deb) for this version is not an option either which makes even installing ROS on this platform a bit more difficult. Even still, diamondback was successfully installed by compiling from source, but it became apparent rather quickly it wasn't going to work correctly. For example, none of the sensors that were tried worked properly with the software. When attempting to get support from the ROS community, problems kept getting traced back to the version of Ubuntu used and incompatibilities with the ROS packages arising from that fact.

Switching over to the PowerBot solved most of these problems as the hardware is much more sophisticated and up-to-date. The only tradeoffs are that this platform takes up a lot of space and is very expensive. The robot was able to remain in the ANT center, which made the size a non-issue, however, if a lot of navigating and driving it around were necessary this would have been much more problematic. Furthermore, based on its high cost, there aren't nearly as many people and institutions working on the platform which made getting support specific to the PowerBot much more problematic. This kind of support is much more readily available with more commonly used robots, such as the Pioneer 3AT or the PR2 robot.

Perhaps the most glaring issue is that ROS is platform dependent in terms of the host OS, requiring the use of Linux. It doesn't stop there in terms of being limited as it requires a specific distribution of Linux (Ubuntu) to be fully supported. Furthermore, ROS's requirements are even more narrow as the newer versions are limited in terms of the release that must be used, with the current pool represented by only four (two when this research began). That said, with the proper release on the right distribution of Linux, ROS was relatively easy to install and configure. The most current ROS builds (electric and diamondback) on various machines under all four fully supported Ubuntu releases were tested and, although getting things to work exactly as they should was somewhat challenging, ROS worked pretty well on all of them.

One final area of configuring ROS that should be addressed is that of ROS packages. It is one thing to get ROS up and running on a machine with all of the basics but inevitably there will be devices that need to be used that require their own custom packages. ROS provides very good examples of how to do this from scratch, however, most of the common sensors out there have already seen widespread use with ROS. Therefore, there is an extensive range of software that is freely available which can be pulled directly from a repository and used out of the box in a COTS-like fashion. Table IV.2 provides a quick overview of the basic structure of the ROS file system.

Table IV.2: Overview of Filesystem Concepts [37]

Packages	Lowest level of ROS software organization. They can contain anything: libraries, tools, executables, etc.
Manifest	Description of a package. Its most important role is to define dependencies between packages.
Stacks	Collections of packages that form a higher-level library.
Stack Manifest	Just like normal manifests, but for stacks.

The recommended installation of a desired package is at the stack level, which is the basic unit of release [17]. This introduces a bit of bloat to the overall filesystem since many of the packages installed with the stacks are not needed. This really isn't such a big deal unless there were a situation where ROS was installed on a drive where storage space is at a premium. There are also system dependencies and package dependencies that need to be resolved when installing a particular package, even at the stack level. For example, one of the packages desired for this test is the PowerBot package authored by Dr. Marius Muja from the University of British Columbia in Vancouver, Canada. There are 12 package dependencies that need to be satisfied for this to work. In order to ensure these were met required searching for them to see if they were installed (*: /rospack find 'name_of_package'*). If they weren't found, they needed to be installed first prior to installing the PowerBot packages. Often times, and this was no exception, these dependencies had their own dependencies, making the issue recursive in nature. This obviously is quite tedious and took a considerable amount of time; this is in addition to the wasted space brought on by the unneeded packages.

4.5.1 MicroStrain AHRS. As can be inferred from the rest of this document, the AHRS worked particularly well with ROS. Though there was not a Microstrain 3DM-GX3-25 driver or package, the older version for the 3DM-GX2-24 was compatible and able to operate the device in the manner required. It is possible certain advanced features only available with the newer version of the sensor would not work with this older version of software but this wasn't tested.

4.5.2 SICK LADAR. The SICK laser would not work on the PowerBot through ROS no matter what was attempted and a very large amount of time was spent attempting to make this happen. There have been others having the same issue, and, it appears to be specific to MobileRobots's robots in particular. For the PowerBot, there is a laser integration board that is connected to the motherboard within the robot itself which, among other things, provides the serial-to-USB connection between the sensor and the onboard computer. This information is available in both the user's manual and the MobileRobots website [26]. The issues others have had in addition to the information pertaining to the integration board appears to corroborate the assumption that this is an issue specific to MobileRobots. However, further testing using multiple platforms has yielded identical results which does not indicate a problem with one specific environment. These findings are captured in a writeup posted to the ROS answers forum [52].

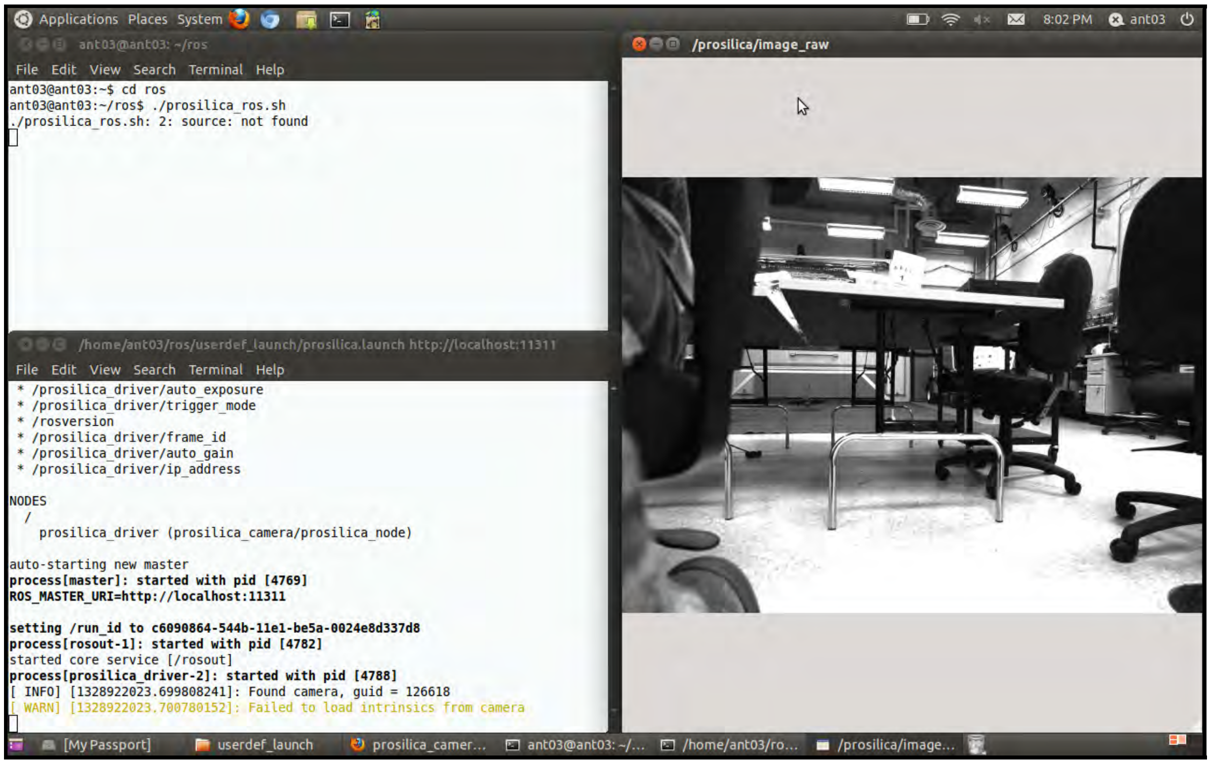
The problem cannot be related to the hardware itself as the device worked perfectly fine using the NextGen VAN software under Player. One thing that worked for another user with a different MobileRobots robot was to change the source code for the driver and recompiling it, adding a delay to allow the data to sync properly [21]. More than likely, there is an issue with the SICK drivers for ROS that has yet to be discovered.

4.5.3 Prosilica Camera. The Prosilica camera also didn't work in ROS on the PowerBot. Tracing down the issue for this particular problem proved quite a bit easier, however. Not only did the camera not work with ROS on the PowerBot, it didn't work using the drivers provided by Prosilica either. Running the Prosilica driver from the command line incited an error preventing the system from crashing and indicating a package inconsistency with Ubuntu. Running the executable through either ROS or graphically by clicking the executable file caused the system to crash, requiring a reboot.

The package problem was never resolved, but, since the camera is small and portable, it was able to be tested on a laptop computer running ROS. Once this was done, everything worked properly and the results were as expected. This indicates a problem is specifically related to the PowerBot and not a problem in ROS. The camera was run in live mode and the results are shown in Figure IV.17.

4.6 Summary

Low-level device handling was achieved but only with specific knowledge of the devices being connected and their interfaces upfront. Scripting was very dependable at getting the software loaded and executing commands autonomously. Even knowing the devices being connected and the location in which they were to be mounted, udev was not



```
ant03@ant03:~$ cd ros
ant03@ant03:~/ros$ ./prosilica_ros.sh
./prosilica_ros.sh: 2: source: not found
```

```
 * /prosilica_driver/auto_exposure
 * /prosilica_driver/trigger_mode
 * /rosversion
 * /prosilica_driver/frame_id
 * /prosilica_driver/auto_gain
 * /prosilica_driver/ip_address

NODES
  /
    prosilica_driver (prosilica_camera/prosilica_node)

auto-starting new master
process[master]: started with pid [4769]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to c6090864-544b-11e1-be5a-0024e8d337d8
process[rosout-1]: started with pid [4782]
started core service [/rosout]
process[prosilica_driver-2]: started with pid [4788]
[ INFO] [1328922023.699808241]: Found camera, guid = 126618
[ WARN] [1328922023.700780152]: Failed to load intrinsics from camera
```

Figure IV.17: Results of running Prosilica with ROS on laptop

successfully utilized for running the bash scripts responsible for configuring and gathering data from them. It may be possible udev cannot be used in the manner intended to produce the expected results. More research and testing in this area would likely bring about the source of complication, however, this would still leave the problem of gathering device information without knowing explicit first-hand knowledge.

The rest of the PnP system is centered on what to do with the sensor and how to handle the data once it is configured. All of the methods presented do this although some of them allow for real-time data processing and device changes while the current working system does not. None of this affects the user of the data except for the manner in which he must integrate with the publisher. That is, if ROS is being used to publish the data, it's probably best that the subscriber is implemented in ROS as well. Still the application software can remain virtually untouched and written according to the agreed-upon interfaces. There are many distinct parts of the system but it can be helpful to look at it in two distinct halves. The first half handles all of the work on the sensor end, keeping track of data that's available and servicing requests. Meanwhile, the second half is responsible for the application of the data, subscribing to the data streams needed to best achieve its purpose and using them to do so.

V Conclusions and Future Work

PnP sensor integration, for N&P along with other applications, has been a subject of high interest over the recent years but has yet to really see fulfilment on a broad scale. Even for a person not particularly familiar with recent trends in sensor applications, a quick web search on the topic makes this all too clear. Unfortunately, even with promising advances from as far back as the turn of this century, IEEE 1451.4 in particular, nothing has really seemed to stick. There have been some claims of success, some particularly recent [10] [14], but these have yet to yield groundswells of advances in this area, at least not yet. The results from the research performed in this study indicate even more promising changes on the horizon as it relates to universal sensor PnP. In particular, people in general are changing the way they are looking at sensors, especially in light of how powerful and accessible they have become. This is a by-product of technology advances in general, where a greater volume of more sophisticated parts are manufactured at drastically decreasing costs [5].

The fact that ROS has arisen and seen such widespread use in quite a short amount of time further reflects a shifting mindset in how sensors and complexity in general are being handled. The whole purpose of ROS being invented and introduced to the robotics community just three years ago was to address the need for better code reuse in a community where applications are quite disparate [36]. One of the major contributors to this complexity is the fact that there are in fact so many sensors being used for a variety of different purposes. It's obvious this need was shared amongst a very large portion of robotics developers. Just glancing at the numbers of packages that have been developed and the number of thread posts on the ROS wiki demonstrates just how pervasive ROS has become in the robotics community.

Observing the capabilities of ROS, what it's being used for, and how quickly it's become embraced by such an eclectic group of individuals, it doesn't seem too much of a stretch that universal PnP for sensor integration is just around the corner. The addition of ROS hasn't taken away from the development of more traditional solutions either; in fact, it's quite possible it has jump-started it. While ROS is focused primarily on robotics applications, many other areas of software development could be greatly improved with a similar type of development.

5.1 ROS Conclusions

There is a considerable learning curve to ROS but it works very well when it is utilized and configured properly. Some of the problems that were encountered weren't specifically tied to ROS and those have been mentioned specifically. It is a bit concerning that ROS is so picky in terms of the requirements of the host operating system. This isn't likely to be an issue for long. To put things in perspective, ROS hasn't been around for very long and the community is definitely aware of the limitations that currently exist. This is one of the reasons they are working to release versions for other operating systems. For example, there is an experimental version for Windows albeit heavily limited in functionality.

The biggest appeal of ROS is that, once it is configured properly and all of the packages are installed, it is capable of providing everything desired in a PnP publish / subscribe system. It provides an environment where sensors can be added and removed without consideration of how it will affect other aspects of the system. Once a sensor is added and creates a node, it is tracked by ROS; the same occurs when it is removed. This is updated in real-time without the need for outside interaction of a user or separate program. Not only does it do this but it does it rather well. In the experiments where multiple ROS nodes were running, adding new sensors and removing them was reflected in ROS immediately. The ROS environment allows for interfacing of outside programs through by subscribing to the ROS topics.

Although ROS was developed for robotics, it can really be used for any application requiring the use of sensors, from one to many. It is plausible to question why one would want to, with all of the other methods and approaches available that can be tailored explicitly and customized for a specific application. The question itself would imply a mindset inconsistent with software reusability. Granted, in a made-from-scratch, novel approach specialized for solving a unique problem, there is much more control over how the software performs as it is designed to strict specifications. However, this control comes at a cost, and that cost can grow exponentially in terms of future maintenance and development costs. For instance, it would be quite difficult to later use some of the same code in another application that is needed in the future; the reason being the software wasn't written with this in mind. The more projects conducted with overlapping requirements and no capability for code reuse very quickly begins to rack up sizeable losses in opportunity cost.

ROS isn't the only way to ensure software reusability, so that again brings to light the question of motivation for using ROS outside of robotics applications. The answer is really quite simple: support. ROS has seen so much use recently that the community of users and developers has grown considerably. The ROS community represents a pool of knowledge and experience for using sensors in a reusable way that is unrivaled by typical COTS software development approaches. Looking at N&P applications as an example, one would find it difficult to find a common COTS sensor that might be needed for a particular testbed that doesn't have a package already available through the ROS community. These packages nearly always include drivers and often contain documentation, launch files, examples, publishers, subscribers, virtual models, etc. If this isn't enough to make it worthwhile just based on the possible savings in development cost, there is also the support community of users and developers that are accessible free of charge as ROS is part of an open source community.

5.2 Scripting and udev

Getting sensors to work autonomously upon being plugged into a system with a host OS is achievable through udev and scripting, however, there are limitations. For one, to use udev it is required the host OS is using the Linux kernel. Secondly, working with serial and ethernet devices are a bit challenging since there isn't usually a handshaking that takes place between the device and the OS once the device is connected to the port. For instance, getting udev to work on a serial port would require active polling, which was addressed earlier.

Another aspect of scripting and udev that represents at the very least a significant challenge is that of permissions. In order to execute scripts in a manner similar to what was done in the tests performed, the ports need to be configured prior to inserting the device. Their default state usually isn't sufficient. For example, the USB ports are configure by default for read and write access but not execution. Serial ports are typically closed by default and must be opened before a serial device can be mounted to one of them. Network (ethernet) ports that are sitting idle do not have an address associated with them by default and must be assigned one prior to use with a device with this type of interface. Scripts often need to run commands that require superuser privileges which is a consideration that needs to be made prior to writing them.

5.3 Traditional Approaches Using Design Patterns

The good thing about the traditional approaches is that, as mentioned previously, there has been some success using them for PnP. There has yet to be a universal PnP architecture successfully built around ROS, at least not yet. The main problem with a traditional SE approach, such as CBSE or OO programming using patterns is complexity. A lot of the coding is pure scaffolding used to create the patterns themselves. Tracing through the code is often a long and arduous task due to the numerous pointers that are often used and the interfacing of numerous software files comprising the complete system. This is where UML models and diagrams are helpful, however, they do not reveal everything. For example, certain lines of code in a particular file may indicate where another file is being used and its location in the overall software suite; this cannot be inferred from a model or diagram.

The complexity of using a traditional approach requires a significant amount of time and resources dedicated to not only writing all of the various parts of the software, but also linking them. A good model of the system upfront helps to counter some of this but creating the model takes considerable time as well without a team of experienced software developers. This is one of the reasons patterns were developed as they allow development to begin with a logical structure aimed toward solving a general problem [2]. In the component-based PnP study, the Observer pattern was leveraged to satisfy the data handling needs of the proposed system.

Interestingly, the Observer pattern cannot solve all of the publish-subscribe requirements needed for the desired effect without significant modifications. For example, the Observer detects state changes, and in this case most naturally to data streams for a particular sensor. What the Observer does not provide, however, is a good way to account for a growable range of sensors. For example, the Observer requires a pointer back to each of the objects it is observing which makes it difficult to keep track of what changes are being generated by which object [31]. There are workarounds for this, but of these, there are many which begin to deviate significantly from the broad overarching structure that makes the pattern desirable in the first place.

5.4 Achievements and Limitations

Partial functionality of a limited PnP system was obtained using ROS as middleware. Scripts were written to automate configuration tasks normally requiring user interaction, while udev rules are still being worked on to make them run whenever the associated device is plugged into the system. The udev rules have also shown partial success, which is reflected in error messages found in the system log files, indicating the rule attempted to run the script it was supposed to once the AHRS device was mounted to the system bus. Furthermore, an analysis was conducted, comparing the feasibility of traditional software development approaches using design patterns against a COTS oriented approach using ROS as middleware.

Testing for the three sensors was conducted on two platforms, the MobileRobots PowerBot and a Dell Latitude E4300 laptop. Diamondback was used as the primary ROS build for both platforms. The PowerBot ran Ubuntu 10.04, Lucid Lynx for its host operating system and the laptop ran Ubuntu 11.04, Natty Narwhal. Table V.1 shows the two platforms and the three sensors, indicating what worked on each.

Table V.1: Sensor Results by Platform Using ROS as Middleware

	MicroStrain AHRS	**Prosilica Camera**	**SICK LADAR**
PowerBot	Y	N	N
Dell Laptop	Y	Y	N

As can be seen in the table, the SICK LADAR did not work on either platform. A great deal of work was put into getting the device to work using ROS, however, the issues were never fully resolved. The problem on both platforms is that after the drivers are executed, the sensor is brought online, and the data stream is started, the device begins to miss scans. It does this repeatedly until it crashes or the process is killed by a termination command.

The other PnP issues that were limited pertain to self-describing behavior. The sensors that were used are passive devices, so they do not provide any information about themselves natively. This required a workaround, essentially utilizing information gathered about the sensor *a priori*. Adding a microcontroller or FPGA to each sensor, which would allow the chip to be programmed to provide self-describing behavior and auto-execution of drivers, is another possible workaround that was not implemented or attempted in this effort. Lastly, udev rules were written for the sensors but were not able to execute the scripts as expected once the devices were mounted to the system bus.

For networking, ROS provides a native publish / subscribe message passing scheme that worked well for the MicroStrain AHRS and Prosilica camera. Each sensor is executed as a *rosnode* and publishes its data to a topic called *rostopic*, which acts very much like a Mediator. It tracks all of the sensors currently publishing data, and is capable of servicing any subscription requests made according the the ROS interface. This allows data to be subscribed to directly in real-time, provided ROS is being used as middleware.

5.5 Future Work

There are two separate paths that can be followed which would benefit tremendously from future work. One follows the ROS approach, using it as a middleware for handling known sensors and their data. This approach will obviously remain a Linux only solution but holds the capability of yielding the biggest gains with the least amount of work. Most of the work in this approach would focus on emulating self-describing behavior when there is none and discovering and configuring sensors once they're connected. The second track would be platform independent and follow a more traditional SE approach. The main area of study would involve further research into the effectiveness of design patterns; most of the work would center around managing sensors and real-time data handling.

5.5.1 ROS-centered approach. The data handling aspect of ROS works very well and it seems to be a good solution for the system that can be implemented straight out of the box. The part that hasn't worked so well is resolving the self-describing behavior and automated script execution needed for a true PnP solution. If this could be resolved fully, and someone became proficient in writing subscribers to ROS publishers, a full PnP solution utilizing ROS as the middleware would be achieved.

As was indicated in the results, the tests conducted relied upon knowledge of the sensors and where they should be mounted in order to get the scripts to work properly. Furthermore, the problem relating to discovery of serial ports was not addressed as this problem was accounted for by using information about the USB converter the serial device was connected to detect it. This is a major limitation to a PnP system aimed at being universal. The use of system tools and a helper script running continuously in the background performing active polling of open ports is an area worth looking into as a means to gather information about devices being connected to the system. This will not remove the need for obtaining ROS packages for any perceived sensor to be used but this isn't really avoidable as the PnP system is design-time static. The same is true for the scripts as they are linked with using those packages.

The scripts should be improved, however, as bash scripting is very powerful and can be employed much more effectively than it was for the tests performed. For example, the scripts can be linked with other scripts, are capable of utilizing conditional logic which can be used to broaden their functionality and can be executed with superuser access. Tying all of these together, a script could exist that configures a specific port; this requires superuser access. That script might be linked by another script that will interface with the ROS packages. In order to link to the ROS packages, the script must know what type of device it's using so conditional logic can be added to select from several possibilities based on the location of where the script was mounted.

The problem related to devices connected over serial port stems from the fact that those sensors are purely passive devices. Outside of active polling, other methods for gathering information about the sensor should be looked into. For example, there may be a way to force self-describing behavior by relating the passive device with something capable of active identification, such as an FPGA or microcontroller. This is similar to the approach taken with AFRL's space-based PnP. Jammes, Mensch, and Smit indicate something like this as well for what they call "dumb" sensors in their idea for a "Residential Device Controller" or RDC [15].

5.5.2 Patterned Approach. The effectiveness of design patterns could be tested much more vigorously, creating all of the components of a possible PnP system using a CBSE approach like the one proposed in Chapter 3 using patterns. This wouldn't have to be component-bases as and OO approach may be easier to implement. In this approach, the main focus should be on the data handling aspect of a PnP system whereby changes can be detected in real time and data can be subscribed to or unsubscribed from at any given time. This requires a way to actively monitor the state of all connected sensors and detect when they are added or removed from the system. This would provide applications with up-to-date information on what data streams are currently available at any given time. Not only does this allow an application to subscribe to a new data stream once it becomes available, it also prevents it from attempting to subscribe to one that is no longer valid, such as when a sensor has gone offline or has been removed from the system entirely.

One of the big pluses to this line of testing is a result of the work already conducted by Jared Kresge in the ANT center in his NextGen VAN data collection software which follows the GoF design patterns. These patterns could be reused, or his software could be modified, to allow for real-time use of the sensor data as opposed to post-processing. The downside of the approach knowing that even if a system were developed that provides the desired data handling, it would either have to be significantly better than what ROS is capable of providing, or fulfill a major requirement that ROS doesn't, in order to be worthwhile from a practical standpoint. Even then, the issue of self-describing behavior and automated execution would still need to be resolved as well.

5.5.3 Summary. This research focused the realization of a universal platform for PnP sensor integration, primarily centered upon the subject of advanced navigation. The overall effort investigates what a PnP system would look like, highlighting key elements and two primary approaches for obtaining them: 1) A more traditional, pattern-based approach and 2) A COTS-oriented approach using ROS as middleware. The result of this feasibility study provides core insights into how a true PnP sensor integration environment could be achieved. It also demonstrates a good amount of functionality which previously did not exist in terms of utilizing multiple sensors in a scalable and user-friendly manner.

This research area is one that certainly would benefit from future work in some of the areas addressed throughout the document. The results indicate a PnP sensor integration environment is obtainable, even for passive sensing devices that include no native self-describing behavior.

Appendix A: Code

1.1 Div / Mod Example [40]

```
#include <iostream>
#include <vector>
using namespace std;

class Subject {
    // 1. "independent" functionality
    vector < class Observer * > views; // 3. Coupled only to "interface"
    int value;
  public:
    void attach(Observer *obs) {
        views.push_back(obs);
    }
    void setVal(int val) {
        value = val;
        notify();
    }
    int getVal() {
        return value;
    }
    void notify();
};

class Observer {
    // 2. "dependent" functionality
    Subject *model;
    int denom;
  public:
    Observer(Subject *mod, int div) {
        model = mod;
        denom = div;
        // 4. Observers register themselves with the Subject
        model->attach(this);
    }
    virtual void update() = 0;
  protected:
    Subject *getSubject() {
        return model;
    }
    int getDivisor() {
        return denom;
    }
};

void Subject::notify() {
  // 5. Publisher broadcasts
  for (int i = 0; i < views.size(); i++)
    views[i]->update();
}

class DivObserver: public Observer {
  public:
    DivObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        // 6. "Pull" information of interest
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " div " << d << " is " << v / d << '\n';
    }
};

class ModObserver: public Observer {
  public:
    ModObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " mod " << d << " is " << v % d << '\n';
    }
};

int main() {
  Subject subj;
  DivObserver divObs1(&subj, 4); // 7. Client configures the number and
  DivObserver divObs2(&subj, 3); //    type of Observers
  ModObserver modObs3(&subj, 3);
  subj.setVal(14);
}
```

Appendix B: Use Cases

This section of the appendix contains all of the use cases for a PnP system in fully-dressed form. These use cases are supplementary to, and overlap, those introduced in Section 3.5. The first ten use cases are required by a proposed system, regardless of whether or not ROS is used as middleware. The last use case, UC-ROS11, is specific to ROS; it basically encompasses the functionality of the Comm Controller, the Driver Handler and the Observer. Essentially, it is an << *includes* >> of UC-04 through UC-06 and UC-08. This reflects the ROS PnP Use Case System Diagram shown in Figure III.5.

Sensor PnP Use Cases

Version 1.20

Revision History

Date	Author	Description of change
11/21/11	Dan Elsner	Consolidate dwarfed use cases
01/16/12	Dan Elsner	Edited use cases and scenarios
02/28/12	Dan Elsner	Separated PnP System from ROS PnP System

Use Case: Add Sensor

Id: UC- 01

Description

User adds new sensor to the system. The process involves a user introducing the sensor into a system using the interface native to the sensor. This is a primary use case since it represents the basics of the entire goal of the project.

Level: User Goal

Primary Actor
User

Supporting Actors
Host OS

Stakeholders and Interests
Comm Controller – Will need to discover the new sensor
Driver Controller – Will need to configure the new sensor for use.>

Pre-Conditions
Host OS must be booted and running on the system.

Post Conditions

Success end condition
Sensor is powered and/or plugged in to the system via its native interface.

Failure end condition:
Device doesn't power or startup script fails.

Minimal Guarantee
None. User must troubleshoot hardware malfunctions.

Trigger
A user desires to use another sensor within his positioning algorithm.

Main Success Scenario

1. User attaches sensor to system using default interface
2. Sensor is powered
3. Host OS mounts sensor to system bus

Extensions

1a. In step 1, the user cannot connect the sensor
   1. User checks to see if interface connection is possible in given system
   2. User retries or aborts depending on 1a.1.

2a. In step 2, the sensor doesn't power up
   1. User checks whether or not an additional power source is required

3a. In step 3, the sensor does not attach to the system bus
   1. User checks whether or not the port is active and properly configured

Variations

2'. Sensor is powered ahead of time by user with a dedicated source

Frequency: As needed according to user requirements

Assumptions

Host OS is capable of detecting hardware changes

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Remove Sensor

Id: UC- 02

Description

<User removes sensor from the system. The process involves a user disconnecting the sensor. This is a primary use case since it is basic within the entire goal of the project.>

Level: User Goal

Primary Actor
User

Supporting Actors
Host OS

Stakeholders and Interests
Comm Controller – Will need to gather information about the hardware

Pre-Conditions
Sensor must currently be configured and running within the system prior to disconnecting.

Post Conditions

Success end condition
Sensor is disconnected and the system recognizes the hardware change.

Failure end condition:
Device cannot be disconnected.

Minimal Guarantee
None. User must troubleshoot hardware malfunctions.

Trigger
A user desires to take a sensor offline for whatever reason.

Main Success Scenario

1. User disconnects the sensor from the system according to the default interface
2. Host OS detects hardware change
3. Comm Controller detects hardware change

Extensions

<Enter Extensions and their steps here>

1a. In step 1, the sensor cannot be disconnected
   1. User waits for the right condition to disconnect the sensor safely

3a. Comm Controller does not detect hardware change
   1. Comm Controller continuously polls for hardware change until detected

Frequency: As needed according to user requirements

Assumptions

Host OS is capable of detecting hardware changes and will not fail to do so.

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Discover Sensor

Id: UC- 03

Description

Host OS recognizes hardware addition and type of hardware

Level: Sub-Function

Primary Actor
Host OS

Supporting Actors
None

Stakeholders and Interests
Comm Controller – Will need to gather information about the hardware

Pre-Conditions
Sensor must be properly added to the system and must contain self-describing information.

Post Conditions

Success end condition
Sensor is detected by the Host OS

Failure end condition:
Device is not detected

Minimal Guarantee
None. User must troubleshoot hardware malfunctions.

Trigger
A device has been added to the system.

Main Success Scenario

1. System detects hardware change
2. System writes information about sensor to log file
3. System keeps track of all connected hardware

Extensions

<Enter Extensions and their steps here>

1a. In step 1, a hardware change is not detected
   1. Sensor must be reinstalled / checked for agreement to specified interface

3a. System loses track of connected hardware

2. Host OS writes error file

Frequency: Whenever new hardware is added to the system

Assumptions

Host OS is capable of detecting hardware changes and creating log files.

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Configure Sensor

Id: UC- 04

Description

After the sensor has been added and recognized by the Host OS, the driver handler will load drivers and configure the sensor for use using ROS.

Level: Sub-Function

Primary Actor
Driver Handler

Supporting Actors
Host OS, Comm Controller

Stakeholders and Interests
Comm Controller – Will need to monitor SOH

Pre-Conditions
Sensor must currently be added to and recognized by Host OS.

Post Conditions

<u>Success end condition</u>
Sensor is configured for use.

<u>Failure end condition</u>:
Device cannot be configured for use.

<u>Minimal Guarantee</u>
None. A device that cannot be configured will be treated as no device installed.

Trigger
A hardware change has been detected by the host OS and the change is picked up the comm controller.

Main Success Scenario

1. Driver handler is notified by comm controller of hardware change
2. Driver handler requests name and location from comm controller
3. Driver handler executes sensor driver
4. Sensor begins publishing data according to launch / configuration file

Extensions

3a. In step 3, the driver does not exist
   1. Exception is thrown and passed to comm controller

4a. No data is collected
    2. Default message is passed with blank data

Frequency: Whenever new sensors are introduced to the system

Assumptions

Host OS is capable of detecting hardware changes

Special Requirements

Performance
1. N/A

User Interface

2. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Monitor Sensor SOH

Id: UC- 05

Description

Comm controller routinely polls state of sensors in operation. SOH data will include time stamp of check, sensor signal quality, whether or not a sensor is publishing, etc.

Level: Sub-Function

Primary Actor
Comm Controller

Supporting Actors
Host OS
Driver Handler

Stakeholders and Interests
KF Nav

Pre-Conditions
Sensor must currently be configured and running within the system

Post Conditions

Success end condition
Current state of sensor is known.

Failure end condition:
N/A

Minimal Guarantee
Either status information is found or else it is offline

Trigger
Specified intervals, according to requirements.

Main Success Scenario

1. Comm controller checks log files for output and error messages
2. Comm controller resolves state of sensor, time stamp, and errors
3. Comm controller logs SOH data

Extensions

1a. No error or output messages are found

1. Comm Controller logs sensor as offline

Frequency: As specified by application

Assumptions

Host OS is capable of detecting hardware changes. Log files of previous sensor outputs and time stamps are kept in a known location.

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Update Sensor Table

Id: UC- 06

Description
Update table keeping track of sensors currently online and time stamps for last SOH check

Comm controller uses results from SOH check to update the table of online sensors.

Level: <Sub-Function>

Primary Actor
Comm Controller

Supporting Actors
Host OS

Stakeholders and Interests
User – helps determine whether or not to remove a sensor
KF-Nav – needs this information to update positioning algorithms

Pre-Conditions
SOH information must be available.

Post Conditions

Success end condition
Table is updated with most current sensors and their SOH information

Failure end condition:
Table will not be updated properly.

Minimal Guarantee
Table will be updated with most current information available.

Trigger
User-determined intervals.

Main Success Scenario

1. <<includes>> Monitor SOH
2. Comm controller parses SOH data
3. Comm controller formats the data for inclusion in sensor table
4. Data is added to sensor table

Extensions

2a. No SOH data is available or is unchanged

1. Do nothing

Frequency: As needed according to user requirements

Assumptions

SOH data has been collected and will be presented in a known format for parsing

Special Requirements

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Publish Sensor Data

Id: UC- 07

Description
Sensor data is published as new measurements are available

Level: User Goal

Primary Actor
Publisher (Sensor)

Supporting Actors
Host OS
Observer

Stakeholders and Interests
KF-Nav – needs this information to update positioning algorithms

Pre-Conditions
Sensor must be configured and actively collecting data

Post Conditions

Success end condition
Sensor data for a particular sensor will be published.

Failure end condition:
No sensor data is published.

Minimal Guarantee
Sensor data is published in real time.

Trigger
Continuous once sensor is configured and running

Main Success Scenario

1. Publisher publishes continuously, reflecting sensor measurements
2. Streamed data is collected and handled by Observer

Extensions

1a. No data is published
   1. Exception is thrown and sensor is ignored

2a. Null or erroneous data is published
   1. Observer handles data as normal

Frequency: Continuous

Assumptions

Software for publishing the data exists and is available.

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Handle Data

Id: UC- 08

Description
Observer handles data publishing and subscription requests for each sensor

Level: Sub-Function

Primary Actor
Observer

Supporting Actors
Host OS

Stakeholders and Interests
Sensor – Needs to publish sensor data
KF Nav – Application must subscribe to data streams

Pre-Conditions
Observer must exist and be capable of handling subscription requests

Post Conditions

Success end condition
Observer accepts data streams from publishers and provides data streams to subscribers

Failure end condition:
Observer fails to service data requests

Minimal Guarantee
N/A

Trigger
Publish or subscribe requests are received

Main Success Scenario

1. Observer receives data stream publishing or subscription request
2. Observer matches stream to sensor requested
3. Observer saves stream to file or retrieves from file
4. Observer provides data stream to subscriber(s)

Extensions

1a. Observer receives no data requests
  1. Do nothing

2a. Observer cannot match sensor with data stream

        1. Throw exception – data stream cannot be handled
    3a. Observer cannot access file
        1. Check local memory and retry
        2. Throw access exception

Frequency: As needed

Assumptions

Observer has permissions to write and read log files.
Observer is capable of tracking all publishers (sensors).
Observer keeps a list of active subscribers and what data they subscribe to.

Special Requirements

Performance
1. Real-time or near real-time access to data required

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Subscribe to Sensor Data

Id: UC- 09

Description
KF-Nav gathers the results from the published data

Level: User Goal

Primary Actor
KF Nav

Supporting Actors
Observer
Host OS

Stakeholders and Interests
User – Nav solution is dependent upon this process

Pre-Conditions
Observer must exist and be capable of handling subscription requests

Post Conditions

<u>Success end condition</u>
Sensor data for a particular sensor will be received by KF Nav.

<u>Failure end condition</u>:
KF Nav is unable to subscribe to the sensor data.

<u>Minimal Guarantee</u>
N/A

Trigger
As needed, according to the application.

Main Success Scenario

1. KF Nav subscribes to a data output stream for a particular sensor
2. KF Nav receives the measurements from the sensor

Extensions

1a. Unable to subscribe to the output stream.
   2. Throw exception

Frequency: As needed

Assumptions

Software for subscribing to the data exists and written according to proper interface with the Observer.

Special Requirements

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Update KF Algorithm

Id: UC- 10

Description
Update location incorporating new information

Level: User Goal

Primary Actor
KF Nav

Supporting Actors

Host OS

Stakeholders and Interests
User – helps ensure best possible positioning solution

Pre-Conditions
Must know about what sensors are being used and how accurate they are.

Post Conditions

Success end condition
KF filtering algorithm incorporates sensor data.

Failure end condition:
N/A

Minimal Guarantee
N/A

Trigger
User runs positioning algorithm.

Main Success Scenario

1. KF Nav updates algorithm according to data received.

Extensions

1a. No data is received
  1. Positioning estimate isn't affected by this sensor.

Frequency: Whenever new data is received

Assumptions

Software for executing the positioning algorithm is available and running.

Performance
1. N/A

User Interface

1. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Use Case: Handle Sensors

Id: UC- ROS11

Description
Sensors are managed using Robot OS (ROS) as middleware

Level: Sub-Function

Primary Actor
ROS

Supporting Actors
User
Host OS

Stakeholders and Interests
KF Nav – Uses data provided by the sensors

Pre-Conditions
Host OS must recognize and mount sensors, then execute scripts associated with those sensors.

Post Conditions

<u>Success end condition</u>
ROS executes drivers for sensors, configures them, and handles all data and bookkeeping tasks.

<u>Failure end condition</u>:
N/A

<u>Minimal Guarantee</u>
N/A

Trigger
Script invokes ROS directly

Main Success Scenario

1. ROS executes launch files and drivers for sensor
2. ROS executes instances of each sensor as rosnodes
3. rosnodes publish data to rostopic
4. rostopic keeps track of all publishers and subscribers
5. Subscribers interface with rostopic directly to subscribe to sensor data

Extensions

1a. No launch files or drivers exist

1. Do nothing.

5a. Subscribers cannot subscribe to data

   1. Proper interface and request must be verified by application or user
   2.

Frequency: Whenever at least one sensor is part of the system

Assumptions

ROS drivers and launch files exist in packages prior to system use.

Performance
1. Data is handled in real-time for all sensors.

User Interface

1. ROS must be installed and configured on the host system and the system running the application
2. The host OS is currently limited to Ubuntu 10.04 – 11.10.

Security
1. N/A

Bibliography

[1] "udev". URL https://wiki.archlinux.org/index.php/Udev. Accessed 8-January-2012.

[2] Bashir, Omar. "Using Design Patterns to Manage Complexity". *Overload Journal*, (96), April 2010. URL http://accu.org/index.php/journals/1622. Accessed online 13-February-2012.

[3] Bell, Donald. "UML basics: The component diagram", December 2004. URL http://www.ibm.com/developerworks/rational/library/dec04/bell/. Online; accessed 3-October-2011.

[4] Bronsard, Francois, Douglas Bryan, W. Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Ásgeir Ólafsson, and John W. Wetterstrand. "Toward software plug-and-play". *SIGSOFT Softw. Eng. Notes*, 22:19–29, May 1997. ISSN 0163-5948. URL http://doi.acm.org/10.1145/258368.258379.

[5] Bröring, Arne, Krzysztof Janowicz, Christoph Stasch, and Werner Kuhn. "Semantic Challenges for Sensor Plug and Play". James Carswell, A. Fotheringham, and Gavin McArdle (editors), *Web and Wireless Geographical Information Systems*, volume 5886 of *Lecture Notes in Computer Science*, 72–86. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-10600-2. URL http://dx.doi.org/10.1007/978-3-642-10601-9_6. 10.1007/978-3-642-10601-9_6.

[6] Ciancetta, F., B. D'Apice, D. Gallo, and C. Landi. "Plug-n-Play Smart Sensor Network With Dynamic Web Service". *Instrumentation and Measurement, IEEE Transactions on*, 57(10):2136 –2145, oct. 2008. ISSN 0018-9456.

[7] DARPA. "All Source Positioning and Navigation". Broad Agency Announcement, November 2010. DARPA-BAA-11-14.

[8] Dunbar, Michael. "Plug-and-Play Sensors in Wireless Networks". *IEEE Instrument & Measurement Magazine*, 19–23, March 2001. ISSN 1094-6969/01.

[9] Fisher, K. A. and J. F. Raquet. "Non-GPS Precision Position, Navigation, and Timing". *Air and Space Power Journal*, 26(2):24–33, 2011.

[10] Fortier, Andrés, Gustavo Rossi, Silvia E. Gordillo, and Cecilia Challiol. "Dealing with variability in context-aware mobile software". *J. Syst. Softw.*, 83:915–936, June 2010. ISSN 0164-1212. URL http://dx.doi.org/10.1016/j.jss.2009.11.002.

[11] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995. ISBN 0201633612.

[12] Group, Object Management. "UML Resource Page". URL http://www.uml.org/. Online; accessed 27-September-2011.

[13] Huizing, M. "Component Based Development". *Xootic Magazine*, January 1999.

[14] Ippolito, Corey A., Greg Pisanich, and Khalid Al-Ali. "Component-Based Plug-and-Play Methodologies for Rapid Embedded Technology Development". *Ifotech@Aerospace 2005, Arlington, VA*, 1–15. Sep. 2005. ISSN AIAA-2005-7122.

[15] Jammes, François, Antoine Mensch, and Harm Smit. "Service-oriented device communications using the devices profile for web services". *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, 1–8. ACM, New York, NY, USA, 2005. ISBN 1-59593-268-2. URL http://doi.acm.org/10.1145/1101480.1101496.

[16] Jändel, Magnus. *Plug-and-Play Robotics*. Technical Report RTO-MP-IST-099, Swedish Defense Research Agency, 2011. Copyright NATO Research and Technology Organisation (RTO).

[17] joq. "How do I install a missing ROS package?", February 2011. URL http://answers.ros.org/question/9201/how-do-i-install-a-missing-ros-package. Accessed 13-February-2012.

[18] Kaur, Arvinder and Kulvinder Singh Mann. "Article: Component Based Software Engineering". *International Journal of Computer Applications*, 2(1):105–108, May 2010. Published By Foundation of Computer Science.

[19] Kodali, Raghu R. "An Introduction to SOA", June 2005. URL http://www.javaworld.com/javaworld/jw-06-2005/jw-0613-soa.html. Online; accessed 15-October-2011.

[20] Kozierok, Charles M. "Plug and Play", April 2011. URL http://www.pcguide.com/ref/mbsys/res/pnp-c.html.

[21] Krishnan, Aravindhan K. "Unabe to get data from sick laser on Pioneer", March 2011. URL http://answers.ros.org/question/257/unable-to-get-data-from-sick-laser-on-pioneer. Accessed 13-February-2012.

[22] Kroah-Hartman, Greg and Kay Sievers. "udev - dynamic device management", 2010. URL http://manpages.ubuntu.com/manpages/karmic/man7/udev.7.html#contenttoc3. Accessed 11-January-2012.

[23] Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0131489062.

[24] Madhavan, Raj, Rolf Lakaemper, and Tams Kalmr-nagy. "Benchmarking and Standardization of Intelligent Robotic Systems". *Proceedings of 14th International Conference on Advanced Robotics (ICAR 2009)*. June 2009.

[25] MicroStrain. *3DM-GX3-25 Data Communications Protocol: Single Byte Command API*, 2010. Version 1.14 rev. 2011.

[26] MobileRobots. “SICK LMS-200 Laser Integration Board”, November 2011. URL http://robots.mobilerobots.com/wiki/SICK_Laser_Integration_Board. Accessed 13-February-2012.

[27] Mochel, Patrick. “The sysfs Filesystem”, 2005. URL http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf. Accessed 11-January-2012.

[28] mysurface. “create your own time stamp”, January 2007. URL http://cc.byexamples.com/2007/01/26/create-your-own-time-stamp/. Accessed 13-February-2012.

[29] Oracle. “Tutorials and Code Camps - Lesson 8: Object-Oriented Programming”. URL http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/oo.html#what. Online; accessed 27-September-2011.

[30] Orogo, Constantine, Michael Enoch, Donald Flaggs, St. Louis Mo, Constantine Orogo, Michael Enoch, and Donald Flaggs. “Javabased Plug-N-Play (Flight) Control Systems for Responsive Spacecraft”. *Paper No. RS4 2006-6002, 4th Responsive Space Conference*, 9. 2006.

[31] Patje. “Implementing a Subject/Observer pattern with templates”, March 2002. URL http://www.codeproject.com/Articles/3267/Implementing-a-Subject-Observer-pattern-with-templ. Accessed 13-February-2012.

[32] PCMAG. “plug and play — PCMAG Encyclopedia”. URL http://www.pcmag.com/encyclopedia_term/0,2542,t=plug+and+play&i=49389,00.asp. Online; accessed 19-September-2011.

[33] PlayerStageProject. URL http://playerstage.sourceforge.net/. Accessed 28-Feb-2012.

[34] Potter, David. “Smart Plug and Play Sensors”. *IEEE Instrument & Measurement Magazine*, 28–30, March 2002. ISSN 1094-6969/02.

[35] Powell, Brian H. and Dr. Karl Muecke. “Sensor Abstraction in Robotics Software Architectures, Computer Science Meets the Real World”. *Advanced Space Technologies in Robotics and Automation. ASTRA 2011. Eleventh Symposium on*, 1–6. April 2011.

[36] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. “ROS: an open-source Robot Operating System”. *ICRA Workshop on Open Source Software*. 2009.

[37] ROSWiki. “Navigating the ROS Filesystem”. URL http://www.ros.org/wiki/ROS/Tutorials/NavigatingTheFilesystem. Accessed 13-February-2012.

[38] Schroder, Carla. “Manage Linux Hardware with udev”, October 2006. URL http://www.enterprisenetworkingplanet.com/netsysm/article.php/3635686/Manage-Linux-Hardware-with-udev.htm. Accessed 8-January-2012.

[39] SearchSOA.com. “Definition: object-oriented programming (OOP)”, May 2008. URL http://searchsoa.techtarget.com/definition/object-oriented-programming. Online; accessed 27-September-2011.

[40] sourcemaking.com. “Observer in C++”. URL http://sourcemaking.com/design_patterns/observer/cpp/3. Accessed 15-Feb-2012.

[41] Sullivan, John. “Definition: Client/Server”, August 2000. URL http://searchnetworking.techtarget.com/definition/client-server. Online; accessed 15-October-2011.

[42] Sun. “Jini Architecture Specification”, 2001. URL http://www-csag.ucsd.edu/teaching/cse291s03/Readings/jini1_2.pdf. Online; accessed 15-October-2011.

[43] Szczys, Mike. “How to write udev rules”, September 2009. URL http://hackaday.com/2009/09/18/how-to-write-udev-rules/. Accessed 15-October-2011.

[44] TechTerms.com. “OOP”. URL http://www.techterms.com/definition/oop. Online definition; accessed 27-September-2011.

[45] Terry, Michael, Matthew Kay, and Ben Lafreniere. “Perceptions and practices of usability in the free/open source software (FoSS) community”. *Proceedings of the 28th international conference on Human factors in computing systems*, CHI ’10, 999–1008. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-929-9. URL http://doi.acm.org/10.1145/1753326.1753476.

[46] user:Golftheman. “File:Operating system placement.svg”, December 2010. URL http://en.wikipedia.org/wiki/File:Operating_system_placement.svg. Online; accessed 22-December-2011.

[47] W3C. “Web Services Architecture”, February 2004. URL http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. Online; accessed 15-October-2011.

[48] Wegner, Peter M. and Rex R. Kiziah. “Pulling the Pieces Together at AFRL - Space Vehicles Directorate”. *Paper No. RS4 2006-6002, 4th Responsive Space Conference*, 10. 2006.

[49] Wiki, ROS. "ROS/Tutorials/UnderstandingNodes". URL http://www.ros.org/wiki/ROS/Tutorials/UnderstandingNodes. Accessed 13-February-2012.

[50] Wiki, ROS. "ROS/Tutorials/UnderstandingNodes". URL http://www.ros.org/wiki/rosout. Accessed 13-February-2012.

[51] Wikipedia. "Coupling (computer programming)", March 2010. URL http://en.wikipedia.org/wiki/Coupling_%28computer_programming%29. Accessed 13-February-2012.

[52] wkr101. "Errors using sicktoolbox_wrapper on Powerbot", December 2011. URL http://ansers.ros.org/questions/3229/errors-using-sicktoolbox_wrapper-on-powerbot. Accessed 13-February-2012.

Vita

Capt Daniel Elsner Elsner graduated high school in 1995 from Sidney High School in Ohio. In 1996, he enlisted and served as a 3E3X1 at Lackland AFB and Robins AFB prior to entering the Airman's Education and Commissioning Program in 2004. Under this program, he attended The Ohio State University where he earned a Bachelor of Science in Electrical and Computer Engineering and commissioned on June 10, 2007.

Capt Elsner's first assignment as an officer was the AFRL Space Vehicles Directorate at Hanscom AFB, Massachusetts as a 62EXC, Developmental Computer Engineer. While stationed at Hanscom, he earned a Master of Liberal Arts in Extension Studies with a focus in Management from Harvard University in May of 2010.

His next assignment was at Wright-Patterson AFB, Ohio as a student in the Graduate School of Engineering and Management at AFIT. Upon graduation in March 2011, his follow-on assignment will be at the 90th Information Operations Squadron at Lackland AFB, Texas.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-03-2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* August 2010 - March 2012

4. TITLE AND SUBTITLE Universal Plug-n-Play Sensor Integration for Advanced Navigation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Elsner, Daniel L., Capt, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way Wright-Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GE/ENG/12-12
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
This research investigates the potential for Plug-n-Play sensor integration for navigation and other applications. Specifically, the requirements of such a system are outlined and attempts are made to achieve them using two separate systems: one using Robot Operating System (ROS) as middleware and the other using more traditional software design patterns. The end result is not so much a deliverable in terms of software, but more of a feasibility analysis comparing the two approaches.

15. SUBJECT TERMS
Plug-n-Play, Component Based Software Engineering, Sensor Integration, Software Patterns, Observer Pattern, Publish-Subscribe, Object Oriented Programming, Robot Operating System, Robotics, Navigation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			KENNETH A. FISHER, Maj, USAF
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	159	(937)255-3636 x4677 ; kenneth.fisher@afit.edu