ΣΤΡΑΤΙΩΤΙΚΗ ΣΧΟΛΗ ΕΥΕΛΠΙΔΩΝ
Τμήμα Στρατιωτικών Επιστημών

Ε Λ Λ Η Ν Ι Κ Η Δ Η Μ Ο Κ Ρ Α Τ Ι Α
ΔΙΙΔΡΥΜΑΤΙΚΟ ΔΙΑΤΜΗΜΑΤΙΚΟ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΑΚΑΔΗΜΑΪΚΟΥ ΕΤΟΥΣ 2016-17
ΣΧΕΔΙΑΣΗ & ΕΠΕΞΕΡΓΑΣΙΑ ΣΥΣΤΗΜΑΤΩΝ
(SYSTEMS ENGINEERING)
(ΠΔ 96 /2015/ΦΕΚ 163Α'/20.08.2014)

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
Σχολή Μηχανικών Παραγωγής & Διοίκησης

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΑΤΡΙΒΗ

Design and Implementation of the Functional Architecture and ROS-based Control/Logic Software for the "Brasidas" UGS

Διατριβή που υπεβλήθη για την μερική ικανοποίηση των απαιτήσεων για την απόκτηση Μεταπτυχιακού Διπλώματος Ειδίκευσης

Υπό:

ΜΑΚΡΗΣ ΔΗΜΗΤΡΙΟΣ

Α.Μ.: 2014018009

ΦΕΒΡΟΥΑΡΙΟΣ 2017

Η Μεταπτυχιακή Διατριβή του /~~της~~ … Μακρῆ Δημητρίου …….. εγκρίνεται:

ΤΡΙΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

(Επιβλέπων)

Επικ. Καθηγητής Παπαδάκης Νικόλαος (ΣΣΕ) ,……..

Καθηγητής Δάρας Νικόλαος (ΣΣΕ) ,……..

Δρ. Σπανουδάκης Νικόλαος (ΠΚ) ,……..

Η Διατριβή εκπονήθηκε στο ΚΕΤΕΣ υπό την εποπτεία του Διοικητή του ΚΕΤΕΣ, Σχη (ΕΠ) Εμμανουήλ Φραγκουλόπουλου.

ΣΕΛΙΔΑ ΣΚΟΠΙΜΑ ΚΕΝΗ

Έτος 2017

Αφιερώσεις – Dedication

Σὲ ὅλους ὅσους πίστεψαν ὅτι ἡ κατασκευὴ τοῦ "Βρασίδα" μποροῦσε νὰ πραγματοποιηθεῖ, καὶ κυρίως στοὺς συνεργάτες μου, Λγο (ΑΣ) Κο Κατσέλη Παναγιώτη, καὶ τὸν Κο Παράσχο Δημήτριο, γιὰ τὴν ἀπεριόριστη ὑπομονὴ ποὺ ἐπέδειξαν συνεργαζόμενοι μαζί μου...

Εἰδικώτερα, ἀφιερώνεται στὸν Κοσμήτορα της ΣΣΕ, Καθηγητὴ Κο Δάρα Νικόλαο, γιὰ τὴν ἀξιοθαύμαστη ἐπιμονὴ καὶ ὑπομονὴ ποὺ κατέβαλλε γιὰ τὴν ἐπιτυχία τοῦ ΠΜΣ.

To all who believed that "Brasidas" could be realised, above them all to my design team partners, Cpt (AA) Katselis Panagiotis and Mr. Paraschos Dimitrios, for their admirable patience in working with me...

Especially dedicated to the Dean of the Military Academy of Athens, Prof. Daras Nikolaos, for his remarkable tenacity and effort towards the success of the Graduate Courses Program.

ΣΕΛΙΔΑ ΣΚΟΠΙΜΑ ΚΕΝΗ

ΕΥΧΑΡΙΣΤΙΕΣ - ACKNOWLEDGMENTS

Ὁ γράφων ἐκφράζει τὶς εὐχαριστίες του πρὸς τὰ μέλη ΔΕΠ καὶ τὸ προσωπικὸ τῆς Στρατιωτικῆς Σχολῆς Εὐελπίδων καὶ τοῦ Κέντρου Ἔρευνας καὶ Τεχνολογίας Στρατοῦ, γιὰ τὴν πολύτιμη συνεισφορά τους στὴ δημιουργία τοῦ "Βρασίδα".

Ἰδιαιτέρως ἐκφράζονται θερμὲς εὐχαριστίες πρὸς τὸν Δκτη τοῦ ΚΕΤΕΣ, Σχη (ΕΠ) Ἐμμανουὴλ Φραγκουλόπουλο, ἡ παροχὴ ἐκ τοῦ ὁποίου ὁδηγιῶν, συμβουλῶν, καὶ ὑλικοῦ, ἀλλὰ καὶ προσβάσεως στὶς ἐγκαταστάσεις τοῦ ΚΕΤΕΣ, ἀποτελεῖ τὴν πλέον οὐσιώδη συνεισφορὰ στὴν κατασκευὴ τοῦ "Βρασίδα".

Εὐχαριστίες ἀποδίδονται καὶ στὸν Καθηγητὴ Κο Μαυρίδη Νικόλαο, χωρὶς τὴν καθοδήγηση καὶ προσπάθειες τοῦ ὁποίου, ἡ σχεδίαση δὲν θὰ εἶχε ποτὲ προχωρήσει πέρα ἀπὸ τὴν θεωρία.

The author wishes to thank the Professors and personnel of the Military Academy of Athens and the Army Research and Technology Center, for their invaluable assistance in the making of "Brasidas".

Above all, the author acknowledges and thanks the CO of the ARTC, Col. Emmanuel Fragoulopoulos, whose provision of instructions, advice, materiel, and access to the ARTC facilities constitutes the most significant contribution to the "Brasidas" project.

The author also wishes to thank Professor Mavridis Nikolaos, without whose guidance and restless efforts, the project would have never left the drawing board.

ΣΕΛΙΔΑ ΣΚΟΠΙΜΑ ΚΕΝΗ

TABLE OF CONTENTS

ABSTRACT

This thesis presents work done as part of the "Brasidas" Unmanned Ground System (UGS) project. The project aims to design and manufacture a prototype unmanned autonomous ground vehicle (codenamed "Brasidas") for outdoors operations, capable of both autonomous and remote operation. The platform will be configurable over a wide range of missions and with varying operational payloads selectable by the user. The initial design concept intended for a patrol and observation robot, with capabilities similar to those of the TALON SWORDS[1] vehicle, being in addition able to operate autonomously.

Herein will be described the initial analysis and design of the "Brasidas" functional architecture, as well as its evolution over time during the project's development, to the current state it stands at today. The initial design concept will be explained, we will attempt to justify the objectives we were aiming for, and the problems we encountered on the way. The solutions we selected and implemented will also be analyzed and described in detail.

1 https://en.wikipedia.org/wiki/Foster-Miller_TALON

INTRODUCTION

Brasidas (Greek: Βρασίδας) (died 422 BC) was a Spartan officer during the first decade of the Peloponnesian War. Thucydides' characterization of Brasidas suggests that Brasidas united in himself the stereotypical Spartan courage with those virtues in which regular Spartans were most signally lacking. Brasidas was apparently quick in forming his plans and carried them out without delay or hesitation. Furthermore, Brasidas was also an accomplished orator, as recorded by Thucydides.

Brasidas' operations as part of his campaign in Macedonia and Thrace were characterized by the rapidity and boldness of his military movements, as well as by his personal charm and the moderation of his demands towards his opponents. During the course of the winter, Brasidas succeeded in winning over the important cities of Acanthus, Stagirus, Amphipolis and Toroni as well as a number of minor towns.

The Brasidas UGS, not unlike the ancient Spartan general, is an innovative robotic system, designed to operate either autonomously or via remote control, ahead of, behind of, or alongside the main force, in a variety of roles. Its operational capabilities are partly determined by a modular payload system that enables it to efficiently operate day and night, in all forms of military operations, be they surveillance, reconnaissance and target acquisition, minefield clearing and Explosive Ordnance Disposal (EOD), or even simply as asset protection, functioning as a robotic sentinel. This last role is in fact what Brasidas was initially conceived and designed for.

The UGS that was to be named "Brasidas" was conceived as part of a semester-long homework, for the first-semester Robotics class, taught as part of the Systems

Engineering graduate course. Initially, the whole idea revolved around a simple robotic vehicle that could autonomously navigate an area, patroling for intruders and monitoring the area's security. This initial, theoretical concept, eventually grew out of the boundaries of a semester paper, and became the "Brasidas" project. This thesis presents part of the work done within the project, specifically the architecture development and implementation of the robot software.

The assembly, programming, and testing lasted almost two years, but the general concensus is that it was really worth the effort. "Brasidas" may not rival Data from Star Trek™: The Next Generation, but it can certainly be used to fill many operational needs of today. Its most distinctive feature is that it has been constructed using only commercial off-the-shelf (COTS) components, and thus it is a system characterized by a highly competitive price, and ease of component repair or replacement.

The booklet is organized as follows:

Part I: Requirements Analysis and Specification.

Wherein will be analyzed:

– The project objectives we envisioned to achieve,

– The functional requirements established as a result of research (market needs/technology availability/etc.) and scenario-based modeling,

– The non-functional requirements (perfomance, cost, scalability and extensibility, interfaces, etc.) we accepted, and

– The allocation of functional and performance requirements in Phases (consecutive operational versions, "Marks" [Mk]), based on the adoption of a spiral development model.

The final products at the end of Part I are the Requirements Specification and Functional Architecture, as well as the project's expected development Phases.

Part II: Design and Implementation.

Wherein will be described:

– The allocation of the system Blocks to the various project Phases,

– The choice of programming languages (python, C++ if and when deemed necessary) and third-party software libraries and toolkits used,

– The implementation of each functional Block for Phase I, referring to the Physical Architecture (Cpt. P. Katselis' thesis) where necessary for the sake of clarity and completeness, with analysis of the fundamental problems encountered, the potential solutions studied and tested for each problem, and of course the one finally adopted and applied.

The final products at the end of Part II are the final specifications and software requirements (in terms of hardware characteristics) of the prototype. This work will describe the implementation of functions only up to Phase I (Brasidas Mk-I).

Epilogue:

We offer some basic conclusions and observations stemming from the design and implementation process thus far undertaken, regarding the degree to which the initial system requirements were satisfied, and how the divergence affects the viability of the initial design concept. We also discuss on the know-how acquired as part of the endeavor. Finally, we offer insight on the future steps we intend to take to continue the development of "Brasidas" into the next planned Phase, and discuss the project's potential for further evolution, extensibility, and scalability, on a vision of a fully operational robotic weapons system.

Part I

Requirements Analysis and Specification

CHAPTER 1 .
Preliminary Mission Analysis

§1. The Current Situation

Several robotic systems have been tested over the past decade, mostly by the U.S. Army, and other western militaries. Some have been deployed operationally, with mixed results. The most promising such systems, representing the vast majority of those systems in operational use today, are UAVs, more commonly known as "drones" (though the two terms have a different exact meaning), like the Predator, the Heron, etc.

All these systems are remotely operated. Aside from very basic functions, like camera stabilization, closed-loop speed control, flight stability control, etc., which are applications of automatic control systems, with very little interest from the perspective of robotics and artificial intelligence, these drones are entirely remotely operated.

Ground systems are far less featured in operations; indeed when most people hear about a ground robot, they think of the robotic drones operated by EOD units. The older TALON SWORDS system (mentioned in the abstract) was one of the few UGSs deployed by the U.S. Army, in limited field role [1]. The limitation came from reluctance on the commanding officers' part to use a remotely-operated weapons system to seek out and kill enemies.

This is not a thesis about ethics. However, it is this author's opinion that a remotely-operated weapon is no different than a weapon one holds in their hands. Once you decide to use lethal force and take lives, the "how" is irrelevant. Clearly, before we can use robotic weapon systems, the world still has a few non-technical issues to solve…

The lessons learned during the SWORDS's use are summarized in the fact that a remotely-operated weapons system can easily substitute soldiers in guard and base patrol duty. They also permit generally more accurate fire than the average soldier can provide. No robotic system, no matter with what sensors it has been fit, can substitute for in-person reconnaissance, although it is possible for a field commander to get a sufficiently ac-

curate picture of the tactical situation via remote sensing. Thus, a remotely-operated weapons and sensors system provides for worse overall performance than boots on the ground, but exposes friendly troops to far less danger.

An autonomous system (or at least one that would not require constant supervision) is also capable of operating 24/7 with constant perfomance. It is thus well-suited to security and monitoring applications, where round-the-clock security otherwise requires careful administration of shifts, different skillsets, and personnel fatigue, to achieve as close to constant a performance as possible.

In the technical aspect, there have now been developed several specialized sensors to allow robots to survey their environment. Planar and 3D scanning lasers, ultrasonic range finders, stereoscopic cameras, IR-based depth sensors (e.g. Kinect), autofocus lenses for cameras, etc. Of course, all of these have existed for decades, however lately the mainstreaming of robotics have allowed the development of cheap, readily available versions of technology that once used to be the purview of the military. Along with the widespread use of technology have come standards that such technologies now conform to, making new designs easier and more robust than before.

§2. Overview of the Proposed System

The initial idea, on which the whole design is based, is that of an autonomous (or at least semi-autonomous) robotic sentinel. It is based on the assumption that it could be used to efficiently augment the security protocols and personnel of a Military Base or otherwise similar facility. In fact, the initial concept vehicle could as well be employed by any organization that requires monitoring of a location or establishment for security purposes or otherwise, especially during non-office hours (i.e. during the night or weekends). This initial idea forms the Mission Needs Statement, which was then expanded and analyzed in detail, to produce the conceived system's requirements at first, and to make design choices later on during the more advanced project stages.

The vehicle will be of a size comparable to that of a human, so that it can access the

same locations and navigate through the same terrain as the human personnel it is intended to replace or augment. Naturally, it is expected that the vehicle will not have the same movement capabilities as a human, but it is not intend to navigate areas that human personnel would not have to go through anyway.

The system should also be able to function in areas wholly or partially unknown, in a reconnaissance role. In this function it should also be able to operate ahead of or alongside human personnel. In fact, being able to send the vehicle ahead of the human personnel in unknown territory (e.g. to determine if the area is mined or contains other hazards) might be of a higher priority that being able to use it to patrol familiar territory.

2.1. The Mission Needs Statement

The Mission Needs Statement can be summarized as follows: *"A robotic sentinel that can function in non-fully-mapped outdoors environments without constant supervision."*

§3. Mission Needs Analysis

To generate a set of preliminary requirements, the Mission Needs Statement forms the basis, which is then expanded further, adding more details in the process [2]. Simultaneously, research was undertaken on whether similar systems are or have ever been fielded in similar roles, on the means by which such a need is currently fulfilled, and an effort was made to determine whether certain technologies that appear to be required, are available and mature [3]. As it turns out, the Israeli Defense Forces (IDF) are preparing to employ in the field a system of similar principles, filling approximately the same operational niche [4]. Of course the IDF have been using UGSs for years, and they have a implemented a solution that is a bit more practical, if even more costly. A more advanced system, related to [4], is presented in [5].

After several brainstorming sessions, the Mission Needs Statement was expanded, some new ideas were incorporated, and certain perfomance measures were quantified. Additional inspiration was taken from study of already existing similar systems, particularly in terms of performance. The initial conceived vehicle design then, as encapsulated by the Mission Needs Statement, can roughly be seen as possessing the following basic

requirements:

(1) It would operate over any (reasonably flat) terrain,

(2) It would travel at speeds comparable to those of a human on foot (walking-jogging),

(3) It could be fully operated remotely if the need arose. Indeed, some functions should only be usable by a (remote) human operator.

(4) It would possess optical sensors (i.e. cameras) registering in both the visible and thermal infrared parts of the spectrum,

(5) It would be able to go to and from its target operational area on its own, without any operator remote control, other than provision of a basic list of waypoints,

(6) It would be able to operate at least 1 Km away from its control station,

(7) It would operate at full capacity under its own power (i.e. without the need to refuel and/or recharge) for at least 4 hours, and

The above set of initial requirements (mixed functional and non-functional) permits assembly of a (coarse) functional architecture, and in turn production of an initial physical architecture. Analyzing these even further will refine on the initial design and lead to a complete requirements specification [6] [7]. These requirements are quite broad, and represent the intended system through a high-level approach. Decomposing each into progressively simpler, more detailed, lower-level requirements presents not only an analytical challenge, but requires research into the potential technical implementation of each lower-level requirement, to make sure it can actually be implemented with the means and knowledge at the project's disposal.

§4. Constraints

Even at this early stage, attention was brought to several constraints and restrictions that would apply in the project. Some were accepted and imposed by the design team, to keep the design within the ability to realize the implementation. Others was known would simply be imposed upon the project due to foreseeable circumstances. In addition, as the

project progressed into the implementation stage, more constraints were revealed, which were hardly obvious in the first place.

The major constraints will be presented and analyzed here for completeness, since they are immutable and one can only accept them, not modify, refine, or ignore them as needed. As a convention, constraints are numbered with a primary ID of 0.

(0.1) "Brasidas" is a Research Prototype

Manufacturing an operational prototype, that would pass all quality certifications and military standards is not the project's intent. It is understandable that these criteria must all be satisfied if a "Brasidas"-like robotic weapon system is to enter service, but the main focus is in solving the basic engineering problems first, in figuring out what can and cannot be done, what knowledge and know-how will be discovered along the way, and to define a broad roadmap for actually designing and constructing a fully operational robotic weapon system.

For example, it is clear that an operational version would require armor; There is at this stage, no interest in researching armor technologies or their applications in a robotic vehicle. Materials and mechanical engineers have already solved those issues to the degree demanded by the intended use of the system, and their application is straightforward. Therefore, no time needs to be wasted on detailing armor requirements.

A similar reasoning applies to other aspects of manufacturing. The research is narrowed down to solving problems that provide know-how in fields where no off-the-shelf solution yet exists, and where the design team is lacking, such as finding an optimal system architecture (both software and hardware), determining computational and power requirements of such a system, required sensors, and so on. Nonetheless, an effort is being made to satisfy as many standards as possible, from among those typically used in those areas where custom solutions had to be implemented.

This constraint implies that the overall development should follow the prototyping paradigm, but that the prototype should be a throwaway prototype [8]. Attempting to push this into manufacturing an evolutionary prototype is a **huge** risk, due to the

design team's minimal knowledge of and familiarity with production and manufacturing methodology.

In fact, even this choice about creating a prototype was not finalized until after the project had moved well into implementation; when the methodology that would be applied was decided upon, the project's design had to undergone a major revision, and still until the time that this document is written (May 2017), this revision is not complete. The development process will be discussed later.

The summed version of this constraint can be contained in the following statement: *"If a subsystem, component, or part can be integrated into the system through a process clearly defined, already known, and with no customization required, said subsystem, component, or part will not be investigated further, and will be included in the research prototype only if it is required for implementation of the requirements of the Mission Needs Analysis, and permitted by the other constraints."*

(0.2) Adopt the Use of COTS Components to the Greatest Extend Possible

"Brasidas" should use Commercial Off-The-Shelf (COTS) hardware as much as possible. Such material is by definition easy to acquire and almost always available. It is also manufactured in bulk, which reduces the manufacturing and repair costs and delivery times compared to custom, on-request solutions.

Further, "Brasidas" should use open source software, and conform to open standards to the highest extend possible. This enables easy maintenance of code, inclusion of code updates and upgrades at no cost, and complete transference of knowledge, so we depend less on third parties for manufacturing and customizations. Indeed, one can take an open-source piece of software, use it, and afterwards maintain or expand it on one's own, writing and testing own code on top of the initial one, at no cost or need to get any license.

This is a 'soft' restriction, in the sense that it is treated more as a strong guideline, than a hard limitation. Open-source software is, after all, as good as someone made it, without any warranty about its perfomance. It is very convenient to use, but ulti-

mately, if some other proprietary solution is acquireable and satisfies the project needs better than any open-source option, the proprietary solution will indeed be selected.

The summed version of this constraint can be contained in the following statement:

"If a functional requirement can be implemented sufficiently well by open-source software or hardware, or by the adoption of an open standard, then these should be selected in preference to any other proprietary options."

(0.3) Funding and Support is Very Specific

There is basically no cash funding available to support this project. Limited support is in the form of materials made available to us from the Army Research and Technology Center (A.R.T.C - Κ.Ε.ΤΕ.Σ.). Such material, however, is not obtained by ARTC per the project's demands, then passed on to the design team. Rather, ARTC searches through its current inventory of parts not used by any other project, and the design team gets to choose which of the available parts might be of use. Obviously, this complicates matters, since one must sometimes adapt the designed solution to the available hardware, or even scrap certain options altogether, since no hardware is available.

Small expenses (simple, cheap parts) can be covered through personal contribution, and indeed this was done so, but this option is obviously out of the question for expensive parts, like scanning lasers or FLIR cameras. This constraint also affects access to weapons and ammunition, preventing the design team from ever running the necessary tests to integrate such options on "Brasidas". This is a constraint that mostly applies to the project's hardware, not software.

The summed version of this constraint can be contained in the following statement:

"If a required hardware or software part has a cost that we cannot cover on our own, or requires any kind of license, it should be considered not available, and discarded as a potential solution option."

§5. The Physical Architecture

The physical architecture of "Brasidas", and its design and assembly are described in detail in the graduate thesis of Cpt (AA) Panagiotis Katselis. The relevant details will be listed here briefly, for completeness.

"Brasidas" is decomposed into subsystems, each in turn consisting of components. The basis from which this architecture was derived, are the Work Breakdown Structures described in [9] for the Surface Vehicle Systems (in particular the Remote Controlled Surface System) and the Unmanned Aerial Vehicles. The first two levels of the WBS elements for "Brasidas" are listed in the following tree overview of the system.

1. The Carrier Vehicle.
    1.1. Propulsion
    1.2. Transmission
    1.3. System Core (System Processor)
    1.4. Power Grid
    1.5. Communications Grid
2. The Payload

The structure of the payload is highly dependent on each payload's mission statement. A payload module only has to obey the following restrictions:

- 2.α It must receive power from the Power Grid.
- 2.β It must connect only to the Communications Grid, and not directly to any other carrier vehicle component. Communication with other vehicle subsystems must be carried exclusively over the Communications Grid.

3. The Control Station.
    3.1. Telemetry and Video Processor
    3.2. Human-Computer Interface
    3.3. Remote Control Console
    3.4. RF Digital Communications Link

The physical architecture will be called upon when allocating requirements and de-

ciding on any software implementation issues.

The specific physical architecture used on "Brasidas" is displayed in a block diagram at the end of Chapter 8.

CHAPTER 2 .
System Modeling

Before delving into the detailed analysis of the system requirements, and given the rather generalized statements listed in the previous chapter, an effort will be made to produce various models of the system and its use, by employing principles from the scenario-based design methodology [10]. This approach is well-suited to the case of "Brasidas", since it allows the most relaxed approach to designing a system.

There is little to no material on projects of a similar nature undertaken within the context of the Greek Armed Forces, and therefore few directives to be followed, which would help steer design through known and tested procedures and methods. Thus, those procedures need to be determined first, to a certain extend, and that cannot be achieved by adopting a strict, structured desing approach such as the Joint Capabilities Integration and Development System (JCIDS) used by the U.S. DoD for designing, acquring, and fielding a new weapon system. While the JCIDS and similar methodology (see for example [11]) aims to both produce a design conforming to the requirements and minimizing development risks and costs, it still requires clear, concise requirements to begin with, something not available in the case of "Brasidas". "Brasidas" is a research prototype, an experiment if you will, which is intended to help produce not only a testbed platform to develop various subsystems, payloads, and allow generic robotic research, but also a guide, a roadmap as to how to proceed with designing an operational robotic weapon system. It is hoped that the whole process will help identify the problems related with, among others, integrating multiple scientific and engineering disciplines, determining an overall abstract system architecture that can be used as a generic template for future designs, hardware acquisition procedures (including a list of potential hardware suppliers and manufacturers) that reduce acquisition delays and hardware cost, standards to which said hardware should conform, and a host of other aspects.

Thus, since "Brasidas" is more free-form than one might expect, the system design approach must also be free-form, flexible, and able to adapt to requirements and de-

mands as these are uncovered along the way. The approach selected is to adopt a spiral development model [12], where consecutive versions of the system will be produced (called Phases), each one adding more and more functionality over the previous one, functionality that has in the meantime been well-tested and designed based on the new, more concise requirements that (hopefully) have been refined as one of the products of the previous Phase [13]. This also offers the advantage of defining material needs as clearly as possible, thus reducing unnecessary spending of the extremely limited (own) funding.

§1. Scenarios

The earlier development Phases, which start from rather vague and generic requirements, will use techniques from the scenario-based design paradigm described in [10], such as scenarios and use cases, an approach well-suited to uncovering and producing technical requirements through a narrative of the designers' (and the clients') vision of use of the system. In layman's terms, one creates short stories describing how the system would be used, and then sifts through these stories, trying to identify functions and requirements which may not be obvious in the first place.

1.1. Scenario #1: Remote Patrol

"Brasidas" is deployed to patrol a pre-designated, pre-mapped area. The operator loads the area map (which can be a simple geographical map) into the system before the mission starts. The Control Station is positioned in a known location (usually a monitoring or operations center); it may remain stationary throughout the mission, or relocate if needed.

As soon as "Brasidas" boots, it attempts to connect to the "Brasidas" network. Once connected, the platform advertises itself on the network, so other platforms and Control Stations can know it is online.

The operator uses the Control Station to connect to the "Brasidas" network, then retrieves a list of all "Brasidas" platforms online (if any others are operating), selects

the platform and connects to it, then switches it to Teleoperation Mode. The operator proceeds to guide the robot around the area of interest, using the feeds from its onboard sensors and cameras to monitor the area. The robot software uses GPS readings to locate the robot on the map.

If the robot comes across other human individuals (either personnel or clients/visitors/etc., the operator can use the bi-directional voice communication channel to speak to the individuals and receive replies, either to pass warnings, give directions or orders, or for any other purpose. The operator can also disable Teleoperation Mode, leaving the robot stationary at a location, and unresponsive to operator commands. The operator can still view the video feeds while Teleop Mode is disengaged. At any moment the operator can decide to disconnect from the platform, and do as he wishes (e.g. take a break, or pack up the Control Center and move it to a different location). At a later time, the operator can reconnect and reassert control of the robot.

1.2. Scenario #2: Supervised Autonomous Reconnaissance

"Brasidas" is deployed to patrol a pre-designated, pre-mapped area. The operator loads the area map (which can be a simple geographical map) and list of pre-assigned wayponts of the patrol route onto the platform storage before the mission starts. The Control Station is positioned in a known location (usually a monitoring or operations center); it may remain stationary throughout the mission, or relocate if needed.

In this scenario, "Brasidas" acts as a mobile version of the U.S. Army's upcoming Unattended Ground Sensor program [14] [15]. A "Brasidas" element (one robot plus its operator and control station) or squad (two elements) would typically be attached to a mechanized infantry platoon or similar-sized echelon.

As soon as it is powered up and its System Core boots, "Brasidas" enters Autonomous Mode. It loads the map and pre-assigned waypoints of the patrol route, then proceeds to move along the route autonomously. It uses SLAM and path planning algorithms to navigate through the map, continuously updating its pose and calcu-

lated path, as well as details of the area, such as landmarks and obstacles, which are not included in the preloaded map. When "Brasidas" reaches the last waypoint, if the route is closed (i.e. the last waypoint is the same as the first waypoint), "Brasidas" restarts the patrol. If the route is open, "Brasidas" moves backwards, following the waypoints in reverse order.

"Brasidas" also attempts to connect to the "Brasidas" network, if the network is available. Once connected, the platform advertises itself over the network, periodically broadcasting its identity and basic status information.

At any point in time, the operator can power on the Control Center and have it connect to the "Brasidas" network, then retrieve and maintain a list of all "Brasidas" platforms online. The operator can connect to the platform from the Control Center at his discretion. The Control Center then receives the video feeds and telemetry of the platform. While connected, the operator may, if he so chooses, engage Teleoperation Mode. In this case, most of the Autonomous Mode's actions are suspended (preempted), and "Brasidas" acts only under the operator's control. SLAM continues to function (in order to update pose and map), but path planning is suspended.

If at any point during the operation, a suitable target appears on the feeds, the operator or echelon's CO can decide to engage, using whatever payload is installed. Potential options would include direct-fire weapons (Recoilless Rifles, GPMGs, LAWs, etc.) as well as target designation systems for standoff weapons (e.g. a laser designator for Hellfires launched from an AH-64 taking cover behind a nearby hill). Engagement happens only in Teleop Mode, to keep a human in the loop and avoid friendly fire incidents.

While connected (regardless of whether Teleop Mode is engaged or not), the operator can also modify the patrol route waypoints or the map. Modifying the list of waypoints is done on the fly, but modifying the map causes Autonomous Mode to restart, wiping all previous landmark and path history. When the operator disen-

gages Teleop Mode, Autonomous Mode resumes control and the robot continues on its patrol.

When the CO decides reconnaissance is adequate, the operator directs "Brasidas" back to the Control Station, either by entering it as a waypoint and relying on Autonomous Mode, or by switching the platform to Teleop Mode and bringing it back via remote guidance.

§2. Use Case Model

Use cases [16] are abstractions of system use that describe a class of scenarions. Each use case describes one aspect (application) of a system, and typically includes several functions, grouped together in a logical role. Use cases help tremendously in identifying functions and requirements during system design [7], especially when the client-provided requirements or Mission Needs Statement are not very specific.

A use case has *actors*, which are entities external to the system (human operators/other processes or system interfaces/etc.), that interact with it. Use case names use active voice verbs *from the system's point of view* (**not** its operator's). Obviouly, each use case may engage different subsystems of a system; each time, the use case is described from said subsystem's point of view.

While scenarios help immensely to reveal requirements that are not otherwise obvious, use cases are particularly suited to categorizing and grouping functionality when there are multiple actors [7]. Only a few use cases are listed here. For the most part, "Brasidas" interacts only with its Control Center (and both are subsystems of the same system), and the Control Center interacts with its operator, so there can only be so many use cases describing interactions between one external actor (mainly, the operator) and the system. The two most important ones, which represent finalized functionality and are the least likely to need review at a later design Phase, are listed here. A few additional use cases, considering interactions between multiple "Brasidas" platforms, pertain to a Phase of the system very far in the future, and will not be discussed here. The results of these studies are contained in this additional constraint:

(0.4) All Autonomous Mode functions of the "Brasidas" carrier vehicle and payload must not depend on any other subsystem and functionality except those already installed on the platform and/or payload.

2.1. Use Case #1: Connect to Platform

This is a very basic use case, that nonetheless brings out several useful functional requirements, as will be analyzed later. The network architecture (about which thus far no mention has been made) will in part be based on this use case.

Subystem: This use case involves the whole system.

Actors: The operator sitting at the control station console ("Operator").

Initiating Actor: The Operator.

Preconditions: The control station is acivated and its RF Digital Communications Link has established a connection to the "Brasidas" network. The VC²S has retrieved a list of all "Brasidas" platforms that are online and ready.

Postconditions: The VC²S is connected to the selected "Brasidas" platform.

Flow of Events:

Actor Steps	***System Steps***
1. The Operator selects one platform from the list.	
	2. The VC²S establishes an initial connection to the selected "Brasidas". If a connection cannot be established, the VC²S informs the Operator and ends this use case. 3. The VC²S uses the initial connection to retrieve platform-specific parameters, such as data stream availability (number and resolution/framerate of each video feed available, telemetry, etc.) and ports to connect on "Brasidas" to receive those streams. 4. The VC²S signals "Brasidas" to begin streaming data. If the signal is acknowledged unsuccessfully, or no acknowledge is received, the VC²S displays an error message and terminates the use case. 5. The VC²S connects to any data stream discovered in the previous step. The display changes to a new window, that can accommo-

	date display of streamed data. 6. The VC²S begins to stream data from "Brasidas", updating the display in real time.

2.2. Use Case #2: Engage/Disengage Teleop Mode

As far as the operator is concerned, switching mode requires a simple flip of a switch (and it should always appear as simple as possible). However, the system handles this change of state in a bit more complicated manner.

This use case describes what often is called "Manual Override" (MOVRD). This is the deliberate intervention by the system's supervisor in a predefined matter, when the system itself begins to diverge from its predefined or expected operation. In Teleop Mode, "Brasidas" becomes a simple, remotely-operated platform. All other functions are suspended.

This use case actually describes two use cases, one for engaging the MOVRD, and one for disengaging it. However, with the exception of the result, the steps in each case are exactly the same, so they are given as a single use case. When differentiation is required, the individual actions in the flow of events are separated by a slash ('/'), giving the action for 'Engage' before the slash, and the action for 'Disengage' after.

Subystem: This use case is initiated from the control station.

Actors: The operator sitting at the control station console ("Operator").

Initiating Actor: The Operator.

Preconditions: The control station is already connected to the platform.

Postconditions: "Brasidas" is in Teleop Mode.

Flow of Events:

Actor Steps	***System Steps***
1. The Operator toggles the MOVRD switch.	
	2. The Control Station signals "Brasidas" of the state change. If the signal is acknowledged unsucessfully, or if no acknowledge is received, the Control Station displays an error message and terminates this use case. 3. The Control Station notifies the Operator and enables/disables use of the remote control console. 4. The Control Station initiates/terminates a data stream link with "Brasidas", to send ve-

locity commands over it.

§3. Preliminary Software Architecture

Before even beginning to analyze the requirements in detail, and based on the hardware architecture and the scenarios presented in this chapter, one can present a top-level model of the software architecture, shown in Figure 2.1: Preliminary System Architecture.

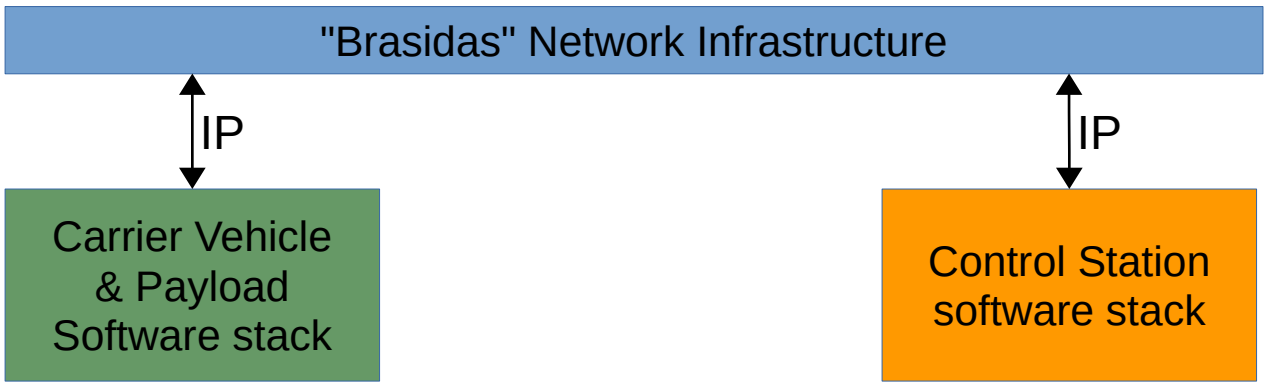

Figure 2.1: Preliminary System Architecture

The obvious pieces of information included in Figure 2.1: Preliminary System Architecture are that each software stack (vehicle, station) operates independently of the other, and that all communication uses the IP protocol stack.

Two questions immediately arise:

a. Should multiple platforms and their control stations be able to connect to the same network, and be visible to one another, as also implied in Use Case #1, or should each robot and its control station communicate over a dedicated link?

b. What technology and communications protocols should the network infrastructure use?

These questions will be tranformed to requirements and constraints during requirements analysis. The answers will come from attempting to select solutions that satisfy those requirements and constraints.

CHAPTER 3 .
Requirements Analysis

Given the abbreviated requirements, the scenarios and use cases, as well as the restrictions listed in the previous chapter, the proposed functional architecture will be analyzed here. First, the initial requirements should be further elaboreated on, always keeping in mind the constraints imposed. It should be emphasized that from this chapter onwards, the primary focus is on the software design and implementation (software components). Any reference to hardware components is made mostly for the sake of clarity, and will be kept brief as needed.

§1. Requirements Determination

Besides the initial list of requirements produced as a result of the Mission Needs Analysis, many more requirements can be extracted from the scenarios and use cases presented in the previous chapter. This task is detailed in this paragraph, separated into subparagraphs related to each item analyzed. The extracted requirements will be simply listed. The next paragraph will analyze each in detail.

We repeat the initial list of requirements here, for completeness. As per the formal standard, each is also assigned a unique serial ID, for easy reference later. From now on, a reference to a numbered requirement will be in the form ***R-XX***, where "XX" is the requirement's numeric ID.

1. Move over Flat Terrain
2. Move at Speeds Comparable to a Human
3. Be Fully Teleoperable
4. Possess Optical and IR Sensors
5. Navigate Autonomously
6. Be Operable at a Minimum Range of 1 Km
7. Operate for at least 4 hours at Maximum Load

1.1. Analysis of Scenario #1

Even before the first paragraph of the scenario description is finished, an obvious requirement is discovered: **Store an Area Map**. This is clearly a functional requirement. A second, related functional requirement appears at the end of the second paragraph: **Track Location via GPS**. This also splits off a hardware requirement – *the platform must have a GPS installed.*

The next discovered function relates to the network infrastructure: **Discover On-line Platforms**. Alongside this is also a non-functional (interface) requirement: **Connect to the "Brasidas" Network**. This last one is required before the system can perform the Discover function, but it is not in itself a system function; rather it is a statement that the Control Station must connect to a network infrastructure first (that will be defined later), before conencting to the platform, and not in an arbitrary manner. Thus, this in effect specifies in an abstract way a form of interface, to which the system should comply. The Control Station must also be able to **Connect to the Platform**, and **Disconnect** from it, without affecting the platform's functionality.

Of course, the platform must also be able to **Connect to the "Brasidas" Network**, and in addition, it must be able to **Advertise** itself (**the Platform**) **on the Network**. Since advertisement happens automatically, regardless of whether other nodes are there to receive, it must happen periodically (not on request). Periodic updating must balance the need to keep the advertised status from going too stale, with the need to minimize bandwidth usage. More on this later.

One more function is the ability for the operator to view the feeds of the onboard sensors and cameras. One of the initial requirements was for the platform to possess optical and IR cameras, but nowhere was it specified that the data from those sensors would need to be available to the Control Station. Though this may appear self-evident, in fact it is not. The sensors could very well be used to feed some onboard algorithm related to the internal operation of the system. Now, reading through the scenario, it is obvious that the system must be able to **Stream Data Feeds**, and also **Stream a Bi-Directional Audio Feed.** In fact, if this is to be used for guiding the robot, then at least one camera (the

Navigational Camera, or ***NavCam***) should be dedicated to this task (at least in Teleop Mode), and its feed should have minimal latency. Thus, an additional, separate requirement is extracted, **Stream NavCam Video with Minimal Latency**. The "minimal" aspect of its latency will be quantified later, during the functional decomposition.

1.2. Analysis of Scenario #2

Since this scenario extends Scenario #1, several functions are common. These will not be listed again, except if additional functionality is discovered.

The first requirement identified is that **Autonomous Mode Must Be the Default Operating Mode**, which makes sense. "Brasidas" is intended to be able to function without a network connection or control station guidance, so when the system starts it stands to reason to enter Autonomous Mode.

The second requirement pertains to how "Brasidas" should follow a defined patrol path. The "again-from-start-if-closed, backwards-order-if-open" approach is in fact nothing more than a simple **Move to Next Closest Waypoint** function. Obviously, since waypoints represent a directed sequence (similar to a single-linked list), the waypoint closest to the current waypoint, besides the previous one from whence we came, is the next in the sequence, unless there is no next waypoint in the sequence, in which case the previous waypoint becomes the next destination.

Autonomous Mode is Suspended in Teleop Mode. This is the next new requirement discovered in Scenario #2, and although it is stated explicitly here, it was until now implied from the manner operator control was portrayed in Teleop Mode – as absolute control. The reason is obvious – having path planning generate motion commands in one way, while simultaneously the operator is trying to steer the robot in a different direction, is a good recipe for disaster. And why does the platform (or payload) need to evaluate potential targets when the operator, who is clearly monitoring the feeds since the system is in Teleop Mode, can identify and classify threats much more reliably? Nontheless, it might be beneficial to have some functionality required for Autonomous Mode operation available even in Teleop Mode (SLAM certainly is). The extend to which Autonomous

Mode functionality is suspended, will be determined during implementation, and in subsequent project Phases.

Target Engagement Happens only in Teleop Mode, at the direct control of the operator. Let's keep the brass happy of moral dilemmas, shall we?

The last two paragraphs also uncover some additional, useful functionality. The operator can modify the waypoint list or the map (**Modify Waypoint List**, **Update Map**), even while Autonomous Mode is active. Of course, if the map is reloaded anew (not simply updated with additional information), which is a different function (**Reload Map**), at the very least path planning must be reset, and the easiest way to do that is to simply wipe the waypoints list (**Wipe Waypoints List on Map Reload**). It's like finding yourself suddenly teleported to a different location; it doesn't matter where you were going, you now need to re-evaluate the situation. Letting the system run unchecked when the map or waypoints are updated is a major breach of reliability, and makes the system behavior totally inconsistent.

1.3. Analysis of Use Case #1

Use cases tend to produce requirements much faster and with less analysis than scenarios, mainly because a use case is already much more structured than a scenario. In addition, the requirements resulting from a use case tend to be more concrete as well. In a sense, the analysis of a scenario done to produce a use case, has stripped away much of the inconsistency, and the results are much clearer.

The main functionality featured in Use Case #1 is the need to standardize the interface of the initial connection. Afterwards, this protocol can be used to retrieve additional functionality specific to each platform. This means the connection process hides the individual platform's complexity and variations, and presents the available functionality in a consistent manner. This is similar to the process one finds in many large frameworks and SDKs (like e.g. DirectX), where a predetermined, always available method provides a common interface that acts as a single point of entry into the functionality of the SDK.

From that interface, the user can call functions to retrieve any additional functionality that may be available on the current platform, but is not guaranteed to be available on other platforms running the same SDK (e.g. framebuffer object access, FSAA, support for DMA transfers, or the like). As an analogy to the architecture elements described in [17], a structurally similar design pattern is the Facade pattern, while the behavior best fits the Mediator pattern.

Thus, so far Use Case #1 results in the following requirements:

Connect to the Platform,

Disconnect from the Platform,

The Initial Connection Must Follow a Common Protocol,

The Functionality Available In the Initial Connection Protocol Must Be Implemented on All Platforms Regardless of Payload, and

The Procedure to Expose Additional Payload-Specific Functionality Must Be Part of the Initial Connection Protocol.

The ability to **Signal the Platform** is another system requirement stemming from Use Case #1. The same requirement mechanism (signaling) appears in Use Case #2, but for a different function (MOVRD, see next section). From Use Case #1, the important aspect of signaling is that it must be acknowledged – it must be a form of two-way exchange. The requirement of acknowledgement also implies need for verification. Signaling (at least within the context of Use Case #1) is used to communicate a request whose complete and correct reception must be communicated back. Depending on the request, its result of execution may also need to be communicated back; **Signaling Must Be Reliable**. Within Use Case #1, signaling is used only to request start of data streaming, but it might have to support additional requests as the implementation progresses, so its implementation should be generalized to be as extensible as possible – **Signaling Must Be Extensible to Support Future Functionality**.

Since signaling will involve acknowledgement, and perhaps will need to support tre-transmission in case the initial communication was not received correctly, it will impose additional overhead in the communication. Overhead means signaling will consume more

bandwidth than its data payload, and will require longer network time to complete, thus a signal is generally expected to have the greatest latency among the various communication protocols that may be involved. The requirement to have minimal latency on the NavCam stream indicates that signaling may not be appropriate for data streaming, but should be used for one-off requests, where completeness and correctness of the communication is preferrable to minimal latency of it.

Finally, the streamed data must somehow be displayed at the Control Station. To the design team this seems self-evident, however what is obvious to one, may be impossible to fathom to another, so the function to **Display Streamed Data** is explicitly mentioned. Note that **All Data Transmitted on a Continuous Basis Should Be Streamed**; platform responses to one-time requests, or propagated events need not be displayed (but should be logged in a logfile or console window). If such cases need to produce displayable data, they should do so by altering the contents of an existing data stream (e.g. telemetry).

1.4. Analysis of Use Case #2

Given what has been said so far, few additional requirements can be extracted from Use Case #2. Most of the functionality described therein has already been encountered and specified in the previous scenarios and Use Case #1. The new functionality uncovered is that the change between Autonomous and Teleop Mode uses signaling, although this seems like an implementation-oriented requirement, and the inclusion of such constraints in the requirements specification should be avoided. Likewise, the use of a separate console, with dedicated controls, during Teleop Mode, is implementation-oriented. Thus, Use Case #2 proves a very implementation-specific use case, and reveals little new functionality. It will not be considered further during requirements analysis, but since it specifies the MOVRD transition, it can be used as a guide during system implementation.

§2. Functional Decomposition

Now that there is a list of requirements sufficiently long and detailed to proceed with designing the system, the next step is to analyze each in detail, and determine how it can be decomposed further.

1.0 Move over Flat Terrain

This is rather obvious. The platform should be mobile, and able to traverse at least the least challenging terrain types, such as flat, rolling terrain. Road mobility is a no-brainer, but ideally the vehicle should possess at least minimal off-road capabilities, making suspension a required feature (suspension is generally not included on robots intended for indoors or on-road movement only).

Still, the intent is not to construct a robotic tank that scales mountains and stairs. Besides not being the vehicle's primary intended operating terrain, a design for such presents significant challenges, best postponed for resolving at a future development Phase, if the need to operate in such terrains becomes more prominent.

2.0 Move at Speeds Comparable to a Human's

A human walking across flat, horizontal terrain, travels at about 3-5 Km/hr, depending on pace. A human jogging moves at between 6 and 10 Km/hr. This is a perfomance requirement. The actual speed chosen in the first version of the hardware specification, which applies to the current development Phase, is 6 Km/hr.

In terms of software implementation, we need not concern ourselves much with how the robot achieves its speed. Motion and velocity commands should be platform-agnostic, and should be translated to signals (voltage and current, absolute values, number of parameters, etc.) specific to the platform used, as late in the command delegation as possible. This enables use of the same software core on multiple, varied platforms, with differing speeds, and only a few scaling parameters need to be changed.

The speed requirement mostly applies to the hardware (physical) architecture. The requirement to have platform-agnostic motion commands, on the other hand, applies

strictly to the software architecture.

2.1 Minimum Platform Top Speed Should Be 6 Km/hr (*performance*).

2.2 Motion and Velocity Commands Should Be Platform-agnostic (*extensibility*).

3.0 Be Fully Teleoperable

The Control Station should have the capability, if the need arises, to manage and regulate all of the robot's functions. Onboard automation should exist to the greatest extend possible, to enhance and augment operator actions, but any and all such automation should feature a "disable" option (dynamic, if possible, without needing to restart or reconnect to the robot). Thus, if the operator wishes to disable a specific function for any reason, they can do so. Obviously, core functionality should be excempt from satisfying this requirement.

4.0 Possess Optical and IR Sensors

To implement this requirement we essentially need cameras, which is a hardware requirement. In software, we need to manage the camera streams, including the NavCam. As a primarily hardware requirement, this will not concern us further in this work.

5.0 Operate Autonomously

When the system starts, the operator will not have connected yet, but it is possible that at least a waypoint list is pre-stored in the system's storage, thus "Brasidas" can and should begin autonomous functions when it boots. If no waypoints are entered, "Brasidas" should not move, at least from a navigational point of view (a future version might feature higher-logic structures that would impose a task to seek cover or explore randomly, mapping the immediate vicinity to a minimum radius, if no navigation path is specified).

The Control Station can connect to the platform and the operator assume control at any point during autonomous operation. Likewise, the operator might relinquish control at any moment, in which case Autonomous Mode must be able to reassert control imme-

diately, with as little overhead as possible. Thus, suspended actions should be preemptable, not requiring a full restart to function again.

Any functionality that is not needed for a quick transition back to Autonomous Mode, but which the operator might occasionally find useful in Teleop Mode, should be made suspendable and preemptable at the operator's control, without its actual state or transition thereof having any effect on subsequent Autonomous Mode or overall vehicle functionality.

When applied to Autonomous Mode functions, this requirement will translate into each function characterized as required to be suspendable or not. For now, this remains a top-level Autonomous Mode constraint. It will be analyzed, evaluated, and defined more specifically during the development Phase of Autonomous Mode.

Autonomous navigation, although just a subset of autonomous operation, is still a major functional requirement, so much so that an entire development Phase is devoted to implementing autonomous navigation. This is a very complex requirement, and spawns an extended tree of lower-level requirements. The current system implementation described herein is of a Phase without this functionality, thus to avoid filling pages with unnecessary details, this requirement will be described here only briefly. Some additional information about the concept of autonomous navigation will be provided in the Epilogue.

Typically, autonomous navigation is broken down into three separate problems:

How to generate a map of the environment, how to localize the robot within the environment, and how to plan a path between two points within the environment.

The first two tasks (mapping and localization) are typically solved by a series of algorithms known as SLAM (Simultaneous Localization And Mapping) – at least that's the trend nowadays. SLAM solves the "chicken-and-egg" reasoning that you need to know your location in order to place objects on the map, and you need to know the position of obejcts on the map in order to determine your location.

Planning a path likewise has another set of algorithms available that can get the job done. These algorithms are typically compute-bound, so whether they can function in real time or not depends entirely on the CPU of the host system.

Sensor readings of the environment are related to the SLAM problem, but not to path planning. All kinds of positional sensor readings, like GPS, IMU, planar laser scans, odometry (wheel encoders), landmark recognition via visual camera snapshots, etc., can be fused together at various levels [18], in order to support a complete SLAM solution. In case the map is preloaded, SLAM must also support the capability to function when the map is reloaded dynamically, even if it is just a restart of the SLAM algorithm.

This requirement can be decomposed into the following two requirements:

5.1 Autonomous Mode Must be the Default Operating Mode (*usability*).

5.2 Autonomous Mode Must be Suspended in Teleoperation Mode (*performance*).

5.3 Navigate Autonomously (*functional*).

5.3.1 Perform SLAM and Pose Estimation (*functional*).

5.3.2 SLAM Must Support Dynamic Map Updates (*usability*).

5.3.3 Follow Path Between Two Points on the Map (*functional*).

This function can further be decomposed into the following, including the additional related functionality uncovered through the analysis of Scenario #2:

5.3.3.1 Plan Path to Target (*fuctional*).

5.3.3.2 Generate Motion (velocity) Commands (*functional*).

5.3.3.3 Path Planning Must Accept Dynamic Waypoint List Updates (*usability*).

5.3.3.4 Path Planning Must Accept Dynamic Map Updates (*usability*).

5.3.3.5 The Waypoint List Must Be Wiped Upon a Map Reload (*reliability*).

One additional parameter, that cannot yet be specified, is how often to generate motion commands (a performance requirement). Needless spamming should be avoided, since it will tax the motor controller, and also flood the serial port and increase CPU utilization. We do know from the motor controller's specification that the watchdog kicks in and stops the motors if no command is received over the serial for 1 sec. Thus, motion commands should be generated at least once

per second. In fact, to account for possible processing delays or otherwise, a safer choice is to generate motion commands at least twice per second.

6.0 Be Operable at a Minimum Range of 1 Km

If a robotic sentinel is to have meaningful applications, it should be able to venture a fair distance away from the Control Station. This is a performance requirement actually (non-functional).

The token distance of 1 Km was selected based on the assumed mission. A sentinel robot should patrol and guard an area of the same approximate size as an Army Base. For most cases, that translates to a range of quite less than 1 Km, but if that same robot is to be used in other, less localized roles, then 1 Km is a decent choice. In a hilly or forested terrain, 1 Km approximately represents the maximum meaningful visual range.

7.0 Operate for at least 4 hours at Maximum Power Load

Assuming "Brasidas" operates in its initially specified role (sentry), it should incorporate seamlessly with a Hellenic Army Base's guard duty rotations. This translates to a patrol duration of 2-3 hrs of near-constant motion. When the shift change occurs, "Brasidas" should also be eligible to be 'relieved' and its shift taken over, either by the next "Brasidas", or by the typical two-man patrol element.

The highest power drain is when the robot is moving (the motors are contributing to the majority of the load). When stationary, the power drain is reduced, as only the elecrtonics are drawing power. To cover the worst-case scenario, where "Brasidas" must constantly be on the move, the Powerplant, which will typically be a battery, should provide at least 3 hrs of functionality at the maximum system load.

"Brasidas" can't simply be switched on and sent on patrol. The robot will likely require a minute or so to boot, the "Brasidas" network will need to be set up, and then the Control Station must connect to the robot and upload an area map and patrol route waypoints, a process that typically takes 10-15 min. Likewise, when the patrol ends, "Brasidas" will need to stay online for a time (typically a few minutes), to allow proper shut-

down. Thus the required minimum operational time must be greater than 3^h20^m. A safety margin of 25% is added to this figure, to account for partial battery charging, battery capacity degradation over the system's lifetime, and other random events, and the minimum operational time ends up at 4 hrs. In fact, the calculations leading to this figure are rather conservative, and the limit should be higher, but for starters it will suffice. This is oviously a performance requirement.

8.0 Connect to the "Brasidas" Network

As shown on Figure 2.1: Preliminary System Architecture, the network infrastructure is a separate functional block. This infrastructure aggregates all the network-specific functionality. Of course, since the network hardware is integrated in part on the platform and in part on the Control Station, the network infrastructure implementation will end up being split likewise. One should also keep in mind that the actual network functionality does not exist as standalone code; while it appears as a single separate functional block, its functions all have the single purpose of allowing the Control Station and platform software stacks to communicate with each other. Thus, some part of the network functionality will be implemented as part of each software stack, each part ending up required to communicate in a specific manner with the part implemented in the other stack.

Therefore, the requirement to connect to the network in the end is not a function that the system must implement, but an interface to which the system must conform. This interface will be defined as part of the network architecture. What is required is conformance to this interface, which makes this an *interface* requirement.

9.0 Discover Platforms Online

After a Control Station connects to the "Brasidas" network, it must somehow figure out the existence of "Brasidas" platforms that are also connected to this network. Likewise, a "Brasidas" robot that joins a network may also need to build a list of other platforms online. Furthermore, this discovery must be ongoing; as long as a node remains connected to the network, it should be able to update its list of other nodes.

As explained in R-16.0 (Advertise Platform Over the Network), discovery should be passive, not active. A note (platform) is responsible for announcing itself over the network. Other nodes should only listen for such announcements, and need not actively send requests to receive node updates. Passive discovery is much more conservative on network resources.

Since discovery is passive, a way must be defined for discarding nodes that were online earlier, but are offline now, and thus no updates are received from them. A node entry in the list should be timestamped with the time of last update received (using the time of the node that maintains the list, not that of the remote node), to facilitate a measure of confidence in whether this node is stale or not. According to R-16.0, a node advertises itself at least once every 3 sec, so to account for clock discrepancies, a list entry whose timestamp differs from the system time by more than 4 sec (stale node), should be considered offline and removed from the list.

This function then decomposes thus:

9.1 Add a Node to the List When Its Status is Received (*functional*).

9.2 Timestamp Discovered Nodes With Time of Last Discovery (*functional*).

9.3 Remove Nodes Whose Timestamp is 4 sec or More Older Than System Time (*performance*).

10.0 Connect to Platform

Assuming a platform has been discovered according to R-9.0 (Discover Platforms Online), an initial connection should be straightforward. This initial connection needs to implement several additional requirements, as explained above in the analysis of Use Case #1. Following that analysis, this requirement can be decomposed as follows:

10.1 Establish Initial Connection (*functional*).

10.1.1 Initialize Network Link (*functional*).

10.1.2 Retrieve Platform-Specific Parameters (*functional*).

10.1.3 Retrieve Additional Platform-Specific and Payload Capabilities (*functional*).

10.2 The Initial Connection Must Follow a Common Protocol (*interface*).

10.3 The Functionality of the Initial Connection Common Protocol Must Be Implemented on All Platforms Regardless of Payload (*interface*).

11.0 Stream Data Feeds

Data streams are continuous feeds of logically grouped information sets. Examples include the NavCam's video stream, another installed camera's video stream, audio stream from the onboard microphone (if one is installed), a stream of telemetry and status reports, etc. A stream may contain raw data (e.g. a video or audio feed), or processed and structured information (e.g. telemetry); thus, this opens the possibility to perform sensor fusion onboard the platform, then stream the resulting information instead of raw data. Depending on the case, this can reduce bandwidth requirements significantly, albeit probably at the cost of additional – and sometimes substantial – CPU utilization.

To stream the sensor feeds over the network produces traffic. Sometimes, this traffic is pointless, such as when no station is trying to receive the feeds. While capturing the sensor feeds onboard might have applications besides streaming them to the Control Station or some other interested node in the network, and therefore can happen regardless of the network state, streaming should only be done when a network request for the feeds has been submitted. Every data stream should be pausable and resumable independently of any other data streams.

Since, as mentioned, streaming increases network traffic, the data stream should be compressed before transmitted, if possible; such a stream obviously needs to be decompressed upon reception before its data content can be utilized. Video feeds can take advantage of any available video compression codec, and the same goes for audio feeds; data feeds are a different matter, but data feeds, even when uncompressed, generally take up a much smaller portion of network bandwidth compared to media streams. No functionality that depends on properly transmitted data should rely on data streams, and should instead use the signaling function.

Thus, this requirement evolves into the following ones:

11.1 Capture Data Feeds from Onboard Sensors (*functional*).

11.2 Determine If a Network Request to Stream a Feed has been Submitted (*functional*).

11.3 Stream the Data Feeds Over the Link (*functional*).

11.4 Each Stream Should be Manageable Independently from Any Other Stream (*usability*).

The NavCam stream follows the guidelines of R-11.0 (Stream Data Feeds), so it is not a separate functional requirement (and receives no separate ID). Its feed however, when the robot is teleoperated, must be transmitted in real-time, or at least near-real-time, since any latency directly affects the smoothness or the robot's responses to guidance, and the operator's response time to obstacles and other events of interest appearing in the camera feed. Ideally, the IR camera stream should have a similar latency limitation, but the IR camera is part of the payload, and its exact performance requirements will be determined along with the other payload requirements.

To quantify the "minimal" amount of acceptable latency, a base calculation can start from R-2.0 (Move at Speeds Comparable to a Human), and take into account the typical reaction time of a human operator, which is around 250 msec. Assume further that the maximum permitted distance the vehicle may move, before the combination of latency and reaction time allow the operator to apply course corrections, is 1 m (positional error). That is, the vehicle's maximum stopping distance should not exceed 1 m. This is also the typical positional error of most modern budget GPS modules. According to the hardware specification, once a stop command is issued, the motors are powerful enough to stop the robot instantly, so there is no distance overhead involved, and the only source of stopping distance comes from the operator's reflexes and network latency.

Then, given R-2.0's speed limitation, the maximum time differential is

$$t_D = \frac{1\ m}{6\ Km/h} = 600\ msec$$

Since a human operator's reflex response time is approximately 0.25 sec, the tolerable latency of the NavCam video feed must not exceed 0.35 sec. We will settle on a value of 0.15 sec, so we can account for the additional latency for a stop command to reach the

robot (round-trip delay), and still have some overhead allowance. This means that this requirement is replaced by the following:

11.5 The NavCam stream must have a latency of 150 msec or less (*performance*).

The bi-directional audio also follows the guidelines of R-11.0 (Stream Data Feeds), so it will be implemented as part of the latter. Latency limitations apply to the bi-directional audio feed as well, although they are far less strict than R-11.5.

12.0 Store Area Map

This requirement pertains primarily to autonomous navigation. The robot should have the capability to store a map of the operational area. At the initial development stage, there is not enough information to specify the exact contents of the map, and the scales (local, global, etc.) it should support. These will need to be defined during another pass along the spiral, when autonomous navigation is being developed.

The Control Station should be able to store a map regardles of the operating mode.

At first glance, the map should store the landmarks used for SLAM. What features constitute a landmark is determined by the specific SLAM algorithm used. Also, the specific path planning algorithm will in turn determine what other information needs to be stored on the map. In addition, the hardware implementation will impose limits on the possible maximum size of the map that can be stored and processed. Finally, the rate at which the map is updated, should be high enough to contain all needed information, but not so high that it imposes unnecessary computational load on the system. For now, this rate limitation cannot be specified further, but even incomplete, it should be included in the specification, if for no other reason than as a reminder to the development team that such a restriction exists and should be determined in a future development Phase.

At this time, this requirement can be decomposed into the following reaquirements only in an abstract approach:

12.1 Load or Reload a Map (*functional*).

12.2 Retrieve Map-Related Sensor Data (*functional*)

12.3 Process Sensor Data to Produce Map-Specific Information Structures (*functional*).

12.4 Update Map (*functional*).

12.5 The Map should be Updated as Often as Needed, and not More (*performance*).

13.0 Track Location via GPS

A GPS module allows each node to determine its approximate location without resorting to more complex algorithms. SLAM generally provides much better positional accuracy than a GPS alone, as it fuses multiple positional sensor data, but SLAM is much more complex and computationaly-intensive, and SLAM will be implemented as part of Phase 2 as per R-5.3.1(Perform SLAM and Pose Estimation). A GPS module can, however, be included since Phase 1, at minimal cost and integration effort, to provide basic localization functionality. Plus, GPS can be used for more than localization, as it includes time information as well (can allow time synchronization).

It should be emphasized that the GPS signal is neither secure nor jam-resistant. The C/A signal available to civilian modules is especially vulnerable. SLAM, which combines multiple sources, is less prone to spoofing.

14.0 Signal the Platform

Signaling is used to send a single request, and get a response. It can be used to implement multiple functions, by properly specifying the signal mechanism. A signal's reception must be acknowledged by the platform, and this acknowledgement is separate from the platform's response to the Control Station's request. A signal is not obligated to know how to handle a response – only to send the request, verify (via the received acknowledgement) that it was received correctly, then take a response and pass it back to the client that generated the request in the first place. This approach ensures that a request and its response are communicated in a reliable manner.

It should be emphasized that at this stage of the design, signaling is unidirectional – from the Control Station to the platform, and not the other way round. However, in a future development Phase a case can be made for signaling to be possible between plat-

forms as well. Whether it would be meaningful in that case to also have signaling from a platform to a Control Station, is a matter that has not yet been studied.

In addition to the possibility of targeting multiple types of receivers in a future revision, a signal mechanism must also be able to communicate any additional signal-dependent functionality that may be introduced in a future Phase. This extendability can be ensured by making the signaling protocol request-agnostic, i.e. not having to know the specifics of individual requests and responses communicated. As long as any additional future functionality can maintain that agnosticism, the signaling protocol should be able to accommodate it without problems.

Signaling should only be possible after an initial connection to the platform has been established, as per R-10.0 (Connect to Platform). As such, the base signaling functionality should be part of R-10.0's Common Protocol specification. Thus, given these guidelines and restrictions, this function can be decomposed like so:

14.1 Send Signal Request (*functional*).

14.2 Receive Acknowledgement (*functional*).

14.3 Receive Response and Return It to Request Initiator (*functional*).

14.4 The Signaling Mechanism Must Be Part of the Initial Connection's Common Protocol (*interface*).

14.5 The Signaling Mechanism Must Be Request- and Response-Agnostic (*interface*).

15.0 Display Streamed Data

The Control Station should be able to display any stream it chooses to request and receive from the platform. "Display" in this case extends beyond the visual aspect. Playing an audio stream over speakers falls under the "display" aspect, as does updating the text of a simple text label component. Each stream will carry its own, additional requirements regarding latency handling.

There is no need to specify in a generic manner how to handle the different stream capabilities of different payloads. While the initial connection to the platform from the

Control Station must follow the same common protocol regardless of the actual options available, the Control Station can have different display implementations, and use the one corresponding to the installed payload, information that the Control Station retrieves *after* it is connected to the platform, as per R-10.1.3 (Retrieve Platform-Specific and Payload Capabilities).

16.0 Advertise Platform Over the Network

As soon as a platform connects to the "Brasidas" network, it must begin to advertise itself. Advertising over a network without prior knowledge of possible receivers means that a platform must broadcast its state, typically by including in the broadcast basic information, such as its ID (which obviously must be unique among all the platforms participating in the same network), location, and perhaps basic status information, most prominently its current playload. Combined with R-0.6 (The Network Must be Based on the IP Protocol), broadcasting under the IP protocol uses the appropriate network address (based on netmask). However, to allow for the possibility to later allow the network to be partitioned into group, multicasting should be allowed alongside broadcasting. Multicasting is not inherently supported in the IP protocol, and instead requires IGMP [19]. This will become a constraint below R-0.6:

0.6.1 The network must support IGMP.

Advertising must be periodic, since it cannot, and should not depend on a request-reply model. A platform or Control Station only knows about other platforms online by receiving their broadcasts. If each node connected to the network were to request a broadcast from every other node every let's say 1 sec, then each network update would result in n^2 advertisement broadcasts and n request broadcasts every 1 sec (because each request would receive n replies). By comparison, a periodic advertisement at the same rate would require only n broadcasts every 1 sec. The bandwidth saved in the second case is obvious.

On the other hand, if the periodic rate is too slow, platform states may end up not being updated fast enough. It makes sense to further limit advertisement broadcasts by

adopting a few criteria about when a platform should broadcast an update. To maintain a reasonably updated tactical picture, this update should happen once every few seconds. A good choice is about 3 sec, so this is the interval at which updated information about a node's state should circulate over the network.

A logical assumption is to require that a platform should broadcast an update if its location changes by at least 1 m (the typical dimension of the current "Brasidas" platform and also the typical accuracy of GPS), or its payload status changes. The same distance-based criterion is used to determine the NavCam's acceptable latency. Based on the results of the latter, the minimum period between broadcasts is no less than 600 msec. The assumed maximum period of 3 sec seems acceptable at the moment; it shouldn't need to be revised until implementation. Thus, we get the following additional requirements:

16.1 Broadcast Platform ID and Status over the Network (*functional*).

16.2 A Platform Must Advertise At Least Once Every 3 sec (*performance*).

16.3 A Platform Must Not Advertise More Often Than Once Every 0.6 sec (*perfomance*).

16.4 A Platform Must Advertise When Its Location Since Its Last Advertisement Has Changed By At Least 1 m (*perfomance*).

17.0 Disconnect from Platform

This one is pretty straightforward. Besides signaling the platform to terminate feed streaming, the platform should switch to Autonomous Mode when the Control Station disconnects, if Manual Override had been engaged. Signaling is described as a separate functional entry, since it is used by multiple other functions, and a requirement that Autonomous Mode be the default mode is specified already under R-5.0 (Operate Autonomously), so there is really nothing more to this function, other than defining a viable transition between modes.

0.5 Target Engagement Happens Only in Teleop Mode

While target identification algorithms might have evolved to be sufficiently accurate

today, the same does not apply to threat assessment and target classification algorithms; these aspects of machine learning and artificial intelligence are still being actively researched, and not mature enough to rely upon them in the field, especially in circumstances as chaotic as a potential battlefield. Thus, any sort of target engagement must be actively supervised and monitored, lest it leads to friendly fire incidents and unnecessary loss of life. This is an overall system operational constraint, and is not limited to the functionality offered by Autonomous Mode.

§3. The Network Architecture

Back in Chapter 2: System Modeling, where an early, rough version of the software architecture was presented (see Figure 2.1: Preliminary System Architecture), two questions were raised with regard to the mentioned network architecture, namely whether multiple "Brasidas" and Control Stations should be part of the same network, and what the network's specifications should be. Now, after discussing in detail the system design and its architecture, it is time to answer these questions. Before doing so, the reader is reminded of the adopted development model: spiral development. This is a case where the network has been implemented at a very basic level, with additional design, development, and testing intended in the future.

3.1. The Initial Version

There is only a single vehicle and a single Control Station, and that leaves very little room to experiment with different network topologies and options. In the beginning, we used a basic case of a point-to-point link, as illustrated on Figure 3.1: The Initial Network Configuration, below.

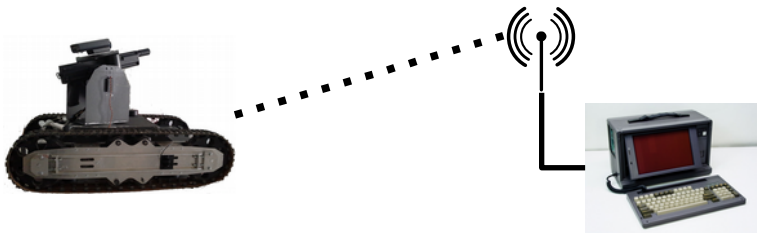

Figure 3.1: The Initial Network Configuration

The only advantage to this limited case, is that one cannot bother with network design along the rest of the system design, even if they wanted to; it lightens the workload, in a sense, and in all honesty, it is not as if there are no other issues to solve during the design process, plus even this trivial solution has proven more than sufficient in the case where only a single "Brasidas" needs to be deployed. Needless to say, however, if "Brasidas" is to ever have a potential for use, this simple network architecture could not stand.

3.2. Specifying the Desired Architecture

Any network specification should basically follow the OSI 7-layer architecture [20]. As a well-studied architecture and open standard, it makes little sense not to adhere to it, especially since virtually every other network systems designer out there does.

The first question posed in Chapter 2: System Modeling, was concerned with whether multiple "Brasidas" platforms should be allowed to interconnect. In the absence thus far of any indications to the contrary, multiple platform interconnection should be allowed. The functionality can always be revoked later if studies show that this is neither required nor convenient. But as a rule, it is generally better to take away a capability that you have available in the first place, that to make available a capability that was absent to begin with.

With respect to the second question, regarding the type of technologies and protocols to use, the given range of R-6.0 (Be Operable at a Minimum Range of 1 Km) hints at the general technology to use: it is obvious that communication should be wireless (unless somehow one finds the prospect of a 1-Km-long cable practical). Since the current solution uses multi-Mbps WiFi and at no point has this link been utilized to over 50%, throughput cannot be quantified in detail at this stage. The greatest amount of link utilization comes from video streaming. Assuming streams are H.264-encoded, an 720p@30 stream consumes around 4 Mbps (average picture quality), and a 1080p@25 consumes around 7 Mbps. Any future solution should provide at least this much throughput at the range required by R-6.0, *per camera feed required*. However, additional overhead may be im-

posed. Network organization structures, and other aspects of communications, touched briefly below, can easily push this single-digits Mbps throughput to a much higher value. All else being equal, it makes sense that the option with the highest throughout should be selected.

For the greatest convenience in system design, the network should follow the IP protocol stack from Layer-3 upwards. The requirements for the Physical and Data-Link Layers will be the subject of this section, but since the whole point of the "Brasidas" network will be essentially to interconnect computers, it is only natural to use the most proven protocol available – the IP protocol. This is the first and most fundamental requirement (constraint, actually) of a desired network architecture:

0.6 The Network Must be Based on the IP Protocol.

As mentioned earlier, there is also a need for nodes to advertise their state over the network, without prior knowledge of other nodes. In general, this requires support of a broadcasting or multicasting protocol on the network's behalf, but since the network must be based on the IP protocol, a multicast protocol (IGMP) is already defined as part of the IP protocol suite (broadcasting is supported by IP itself), so this must be supported as well.

3.3. The Current Situation

The current network architecture is based on the mesh topology described later in this paragraph. Of course, a two-node mesh is rather trivial, but it is enough to test the basic aspects of networking and decide on proper policies (addressing scheme, etc.).

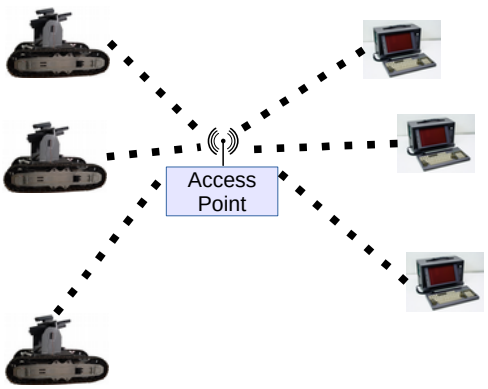

Figure 3.2: Point-to-MultiPoint Configuration

A military-grade, operational network of "Brasidas" platforms and Control Stations would require network management and disruption tolerance. The two prevalent topologies for more than two participants are point-to-multipoint (PtMP) and mesh.

PtMP, or star topology, shown on Figure 3.2: Point-to-MultiPoint Configuration, has a single net-

work controller node (think of WiFi's Access Points), which represents a single point of failure, hardly a disruption-tolerant network. Imagine if, during an operation, enemy fire were to take out this network controller node; the entire network would collapse and all "Brasidas" platforms would be useless. In addition, as the platforms move around, they must all remain within range of the Access Point at all times, or else they lose connectivity. In addition, the AP must simultaneously service all the clients, so unless it employs an advanced MU-MIMO scheme, the effective bandwith drops and latency skyrockets. The only benefit, if the AP is stationary and its location is known, is that Control Stations can then use directional antennas for increased range. Clearly however, **PtMP is not appropriate** as it provides virtually no disruption tolerance or adaptability to changing field circumstances.

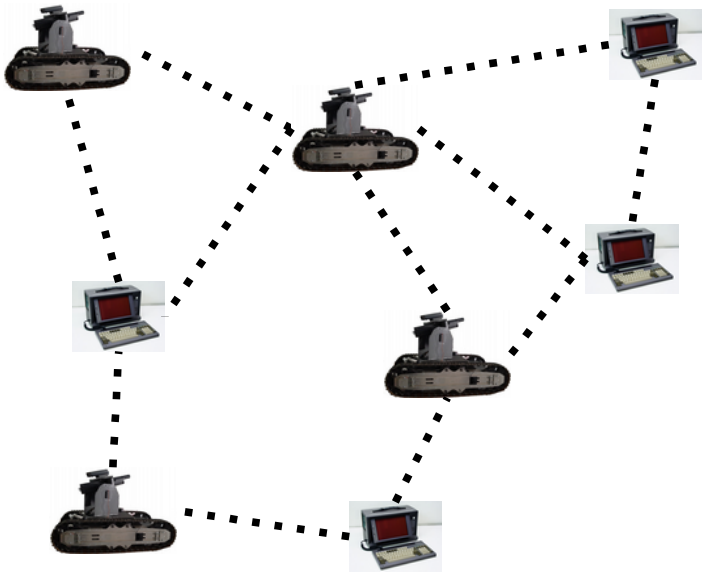

Figure 3.3: MANET Mesh Configuration

A mesh topology on the other hand, in particular that of a Mobile Ad-hoc NETwork (MANET), serves the concept almost too well. Such a possible configuration is depicted on Figure 3.3: MANET Mesh Configuration. Under such an architecture, every "Brasidas" platform or Control Station is treated equally as a node. Nodes can join or leave a mesh network dynamically, and modern mesh routing protocols like OLSR and BATMAN can route packets via the shortest hop path and compensate for nodes leaving the network with great efficiency, making such networks self-healing. As long as each node in a mesh network can "see" at least two others, the network is disruption-tolerant. Such a dynamic network configuration may occasionally present large traffic latencies, however; this is a topic that requires further research. Nonetheless, this culminates in the second and third requirements (again, both constraints) for the desired "Brasidas" network:

0.7 The Network Must be Structured According to the Mesh Topology, and

0.8 The Network Must Implement a Routing Protocol for Multi-Hop Ad-hoc Networks.

Well, in short, the network should just look like that of Figure 3.3: MANET Mesh Configuration. Care should be taken in the case of a MANET that throughput is affected by the number of hops as little as possible; it is well known, for example, that single-radio WiFi repeaters reduce throughput by half, since their radio must switch between receiving and transmitting. Likewise, when a station has to serve multiple client links (the WiFi term is 'spatial stream'), it has to transmit or receive only a specific client's data each time, and ignore the others. In a mesh network, where links are dynamic and change as the nodes move around, such problems can become insurmountable if the circumstances happen to be 'just right', such as when the network ends up being partitioned in two clusters, and only a single node can connect to both. This single node acts as a bridge and a bottleneck at the same time, if it suffers from the shortcomings mentioned in this paragraph.

A typical solution to the problems discussed in the previous paragraph is to use multiple radios per node, typically two radios, one to transmit and one to receive. This solves the throughput issue, but still is unable to handle the one-client-at-a-time issue. In addition, using multiple radios in close proximity produces cross-talk, and causes significant interference. A new technology that greatly alleviates both problems is MU-MIMO, which essentially can handle multiple simultaneous spatial streams, each directed at a different client. At the moment, only a few commercial routers support this option, and only in the 5 GHz band, which is extremely short-ranged.

3.4. A Glimpse of the Future

Range and throughput may be the primary requirements from a functional and design point of view, but they are not the only ones. One should not forget that "Brasidas" is intended to (some day) be a military system, and the only two attributes of military wireless communications that matter are security and jam resistance. Needless to say,

WiFi may be fast and cheap, but it is lacking in security. AES128 found in most commercial WiFi transceivers, is NSA-approved and practically unbreakable, but is nonetheless *not* recommended by NSA for encrypting information classified above TOP SECRET (AES128 is Type 3 and part of NSA's Suite B, but not Suite A), so acceptability of its use depends on the circumstances.

The "official", NSA-sanctioned policy is recommended in [21], under the "Mobile Access Capability Package" heading. However, NSA is preparing an upcoming cryptography suite, to replace the elliptic-curve-based algorithms of Suite B with quantum resistant algorithms.

Security means data should be transmitted over the network encrypted with a "sufficiently secure" encryption scheme. How secure should "sufficiently secure" be, and which algorithms are adequate, is a requirement (constraint) that will be specified by the end user. Even if the end user does not specify the parameters of COMSEC, some form of access control to the network must be implemented. That is the fourth network requirement (constraint):

0.9 The Network Must Implement At Least an Access Control Protocol.

The WiFi modulation scheme is also the joke of jam resistance, since a second Access Point transmitting on the same channel can effectively disrupt communications. Jam resistance mostly means using either FHSS or DSSS modulation schemes, although other techniques, such as CCSS (a coded modulation scheme, see [22] and [23]) are beginning to emerge. Unfortunately, including military-grade communications hardware in "Brasidas" is impossible given constraint 0.3, and besides constraint 0.1 says that as a problem already solved, it can be ignored. Since most communications solutions today are designed based on the OSI 7-layer architecture, security and jam resistance are capabilities that are implemented transparently over the other attributes (security is usually implemented at the session or transport layers), and in the majority of commercial solutions examined briefly do not affect range or throughout. Jam resistance depends entirely on the end user's needs; there is no requirement to provide the "Brasidas" network with any kind of

jam resistance, unless the end user specifies otherwise.

§4. Requirements Overview

Table 3.1: Requirements Overview lists all requirements produced so far, in tree form. It includes the initial requirements, maintaining of course the numbers allocated to each, and is augmented with all the requirements discovered during Requirements Elicitation and Decomposition, each now formally numbered. The requirements are also categorized according to their type, to make allocation easier.

Table 3.1: Requirements Overview

Numeric ID	Description	Type
0.1	"Brasidas" is a research prototype	Constraint
0.2	Adopt the use of COTS components and open standards	Constraint
0.3	Funding and Support is very specific	Constraint
0.4	All Autonomous Mode functions must depend only on installed capabilities	Constraint
0.5	Target engagement happens only in Teleop Mode	Constraint
0.6	The network must be based on the IP protocol	Constraint
0.6.1	The network must support IGMP	Constraint
0.7	The network must be structured according to the mesh topology	Constraint
0.8	The network must implement a routing protocol for multi-hop ad-hoc networks	Constraint
0.9	The network must implement at least an access control protocol	Constraint
1.0	Move over flat terrain	Hardware
2.0	Move at speeds comparable to a Human's	Performance
2.1	Minimum platform top speed should be 6 Km/hr	Performance
2.2	Motion and velocity commands should be platform-agnostic	Extensibility
3.0	Be fully teleoperable	Functional
4.0	Possess optical and IR sensors	Hardware
5.0	Operate autonomously	Functional
5.1	Autonomous Mode must be the default operating mode	Usability
5.2	Autonomous Mode must be suspended in Teleop Mode	Performance
5.3	Navigate autonomously	Functional
5.3.1	Perform SLAM and pose estimation	Functional
5.3.2	SLAM must support dynamic map updates	Usability
5.3.3	Follow path between two points on the map	Functional
5.3.3.1	Plan path to target	Functional
5.3.3.2	Generate motion commands	Functional
5.3.3.3	Path planning must accept dynamic waypoint list updates	Usability
5.3.3.4	Path planning must accept dynamic map updates	Usability
5.3.3.5	The waypoint list must be wiped upon a map reload	Reliability
6.0	Be operable at a minimum range of 1 Km	Performance
7.0	Operate for at least 4 hrs at max power load	Performance
8.0	Connect to the "Brasidas" network	Interface

Table 3.1: Requirements Overview

Numeric ID	Description	Type
9.0	Discover platforms online	Functional
9.1	Add a node to the list when its status is received	Functional
9.2	Timestamp discovered nodes with time of reception	Functional
9.3	Remove nodes whose timestamp is 4 sec or more older than system time	Performance
10.0	Connect to platform	Functional
10.1	Establish initial connection	Functional
10.1.1	Initialize network link	Functional
10.1.2	Retrieve platform-specific parameters	Functional
10.1.3	Retrieve additional platform-specific and payload capabilities	Functional
10.2	The initial connection must follow a common protocol	Interface
10.3	The functionality of the initial connection's common protocol must be implemented on all platforms regardless of payload	Interface
11.0	Stream data feeds	Functional
11.1	Capture data feeds of onboard sensors	Functional
11.2	Determine if there is a network request to stream data feeds	Functional
11.3	Stream data feeds over the network	Functional
11.4	Each stream should be manageable independently from others	Usability
11.5	The NavCam stream must have a latency of 150 msec or less	Performance
12.0	Store area map	Functional
12.1	Load or reload a map	Functional
12.2	Retrieve map-related sensor data	Functional
12.3	Process sensor data to produce map-specific information structures	Functional
12.4	Update the map	Functional
12.5	Map update rate must be optimal (TBD)	Performance
13.0	Track Location via GPS	Functional
14.0	Signal the platform	Functional
14.1	Send signal request	Functional
14.2	Receive acknowledgement	Functional
14.3	Receive response and return it to request initiator	Functional
14.4	The signaling mechanism must be part of the initial connection's Common Protocol	Interface
14.5	The signaling mechanism must be request- and response-agnostic	Interface
15.0	Display streamed data	Functional
16.0	Advertise platform over the network	Functional
16.1	Broadcast platform ID and status over the network	Functional
16.2	A platform must advertise at least once every 3 sec	Performance
16.3	A platform must not advertise more often than once every 0.6 sec	Performance
16.4	A platform must advertise when its location since its last advertisement has changed by at least 1 m	Performance
17.0	Disconnect from platform	Functional

CHAPTER 4 .
Requirements Allocation

The basic architecture to begin with is that of Figure 2.1: Preliminary System Architecture. Initially, the requirements of Table 3.1: Requirements Overview will be allocated to the basic blocks of the initial architecture, then each block will be segmented further as deemed practical given the functional requirements allocated to it.

The allocation of some requirements is pretty obvious; others require some insight, that occurred after the design team traversed the requirements loop multiple times. This multiple-traversal journey will not be described here, only its end results, namely how requirements ended up being allocated to each architecture block. This allocation is presented in brief on Table 4.1: Requirements Allocation. The further development of the architecture, that results from this allocation, is analyzed in the next chapter.

Table 4.1: Requirements Allocation

Numeric ID			Network	Ctrl Station	Platform
0.1		"Brasidas" is a research prototype	X	X	X
0.2		Adopt the use of COTS components and open standards	X	X	X
0.3		Funding and Support is very specific	X	X	X
0.4		All Autonomous Mode functions must depend only on installed capabilities			X
0.5		Target engagement happens only in Teleop Mode		X	X
0.6		The network must be based on the IP protocol	X	X	X
	0.6.1	The network must support IGMP	X	X	X
0.7		The network must be structured according to the mesh topology	X		
0.8		The network must implement a routing protocol for multi-hop ad-hoc networks	X		
0.9		The network must implement at least an access control protocol	X	X	X
1.0		Move over flat terrain			X
2.0		Move at speeds comparable to a human			X
	2.1	Minimum platform top speed should be 6 Km/hr			X
	2.2	Motion and velocity commands should be platform-agnostic		X	X
3.0		Be fully teleoperable		X	X
4.0		Possess optical and IR sensors			X
5.0		Operate autonomously			X

Table 4.1: Requirements Allocation

Numeric ID					Network	Ctrl Station	Platform
5.1				Autonomous Mode must be the default operating mode			X
5.2				Autonomous Mode must be suspended in Teleop Mode			X
5.3				Navigate autonomously			X
	5.3.1			Perform SLAM and pose estimation			X
	5.3.2			SLAM must support dynamic map updates			X
	5.3.3			Follow path between two points on the map			X
		5.3.3.1		Plan path to target			X
		5.3.3.2		Generate motion commands			X
		5.3.3.3		Path planning must accept dynamic waypoint list updates			X
		5.3.3.4		Path planning must accept dynamic map updates			X
		5.3.3.5		The waypoint list must be wiped upon a map reload			X
6.0				Be operable at a minimum range of 1 Km	X		
7.0				Operate for at least 4 hrs at max power load			X
8.0				Connect to the "Brasidas" network		X	
9.0				Discover platforms online		X	
9.1				Add a node to the list when its status is received		X	
9.2				Timestamp discovered nodes with time of reception		X	
9.3				Remove nodes whose timestamp is 4 sec or more older than system time		X	
10.0				Connect to platform		X	X
10.1				Establish initial connection		X	X
	10.1.1			Initialize network link		X	X
	10.1.2			Retrieve platform-specific parameters		X	X
	10.1.3			Retrieve additional platform-specific and payload capabilities		X	X
10.2				The initial connection must follow a common protocol		X	X
10.3				The functionality of the initial connection's common protocol must be implemented on all platforms regardless of payload			X
11.0				Stream data feeds			X
11.1				Capture data feeds of onboard sensors			X
11.2				Determine if there is a network request to stream data feeds			X
11.3				Stream data feeds over the network			X
11.4				Each stream should be manageable independently from others		X	X
11.5				The NavCam stream must have a latency of 150 msec or less			X
12.0				Store area map			X
12.1				Load or reload a map			X
12.2				Retrieve map-related sensor data			X
12.3				Process sensor data to produce map-specific information structures			X
12.4				Update the map			X
12.5				Map update rate must be optimal (TBD)			X
13.0				Track Location via GPS			X
14.0				Signal the platform		X	X
14.1				Send signal request		X	

Table 4.1: Requirements Allocation

Numeric ID		Network	Ctrl Station	Platform
14.2	Receive acknowledgement		X	
14.3	Receive response and return it to request initiator		X	
14.4	The signaling mechanism must be part of the initial connection's Common Protocol			X
14.5	The signaling mechanism must be request- and response-agnostic			X
15.0	Display streamed data		X	
16.0	Advertise platform over the network			X
16.1	Broadcast platform ID and status over the network			X
16.2	A platform must advertise at least once every 3 sec			X
16.3	A platform must not advertise more often than once every 0.6 sec			X
16.4	A platform must advertise when its location since its last advertisement has changed by at least 1 m			X
17.0	Disconnect from platform		X	X

Some requirements (particularly constraints) apply to more than one subsystem. In this case, the functionality contained in the requirement can be split among the subsystems it is assigned to, or it can apply equally to all; the distinction depends on the requirement. Each subsystem's architecture must be designed to accommodate the assigned common functionality or restriction.

Usually, a decomposition of such a requirement will yield functional elements that can clearly be assigned only to one of the subsystems. However, it was deemed acceptable that requirements not be decomposed to such a depth. This work refers to a product not yet complete, and even though the initial requirements have been revised many times, it is almost certain that as development continues, they will be revised even more. To the author's opinion, it would be a waste of time and effort to attempt to specify exactly something that might be entirely different after a while.

For example, in the case of R-10.0 (Connect to Platform), both the Control Station and platform must implement some functionality in order for a connection to occur. The decomposition of R-10.0 does not seem to lift the ambiguousness, so a deeper decomposition should be attempted. However, up until a few months before this writing, the concept of data streams had not even been introduced in the design. Its introduction

changed the connection process, which up until then included no prospect of dynamically differentiated payload options. The possible further development of the network architecture is bound to change the connection process again, since the network used now has no access control method implemented, and likewise connection to a platform implements no authentication scheme. The requirements specified herein are those of the **complete version**, to the extend that they are considered **final**, but are nowhere near implemented completely in the current version. They are not decomposed as extensively as perhaps one would expect, nor are they likewise implemented with finality in mind, to allow revisions, additions, and modifications during the entire development process.

CHAPTER 5 .
The Functional Architecture

Now that a rough initial architecture has beed derived and the requirements have been allocated over its elements, it is time to develop it further.

The elements of the initial architecture will be examined one by one, and each will be further structured accordingly. The requirements allocated to ach element are listed at the beginning of the respective section, for completeness and ease of reference.

§1. The Network Architecture

According to the requirements allocation scheme of the previous chapter, the network must satisfy the following requirements:

Table 5.1: Network Requirements

Numeric ID			Network	Ctrl Station	Platform
0.1		"Brasidas" is a research prototype	X	X	X
0.2		Adopt the use of COTS components and open standards	X	X	X
0.3		Funding and Support is very specific	X	X	X
0.6		The network must be based on the IP protocol	X	X	X
	0.6.1	The network must support IGMP	X	X	X
0.7		The network must be structured according to the mesh topology	X		
0.8		The network must implement a routing protocol for multi-hop ad-hoc networks	X		
0.9		The network must implement at least an access control protocol	X	X	X
6.0		Be operable at a minimum range of 1 Km	X		

These are mostly constraints, stemming from the fact that, as mentioned earlier, the network architecture and development has been deliberately postponed, until other robot functionality more pertaining to the Mission Needs Statement could be developed.

Given the requirements of Table 5.1: Network Requirements, an OSI-based architecture for the network can be derived. Extending R-0.6, there should never be a need to examine network protocols above Layer 3; that is in fact why R-0.6 is there, to allow the use of the many readily available such implementations. Typically, only R-0.9's access con-

trol protocol (EAP on 802.11) will operate at Layer 3 (most likely) or above (unlikely), and there are even options to push it down to Layer 2 (MAC). EAP (implemented as WPA2 over AES) is a good choice here. A mesh routing protocol (R-0.8) such as OLSR also operates typically between Layer 2 and Layer 3. Likewise, a possible future inclusion of a jam resistance protocol would most likely involve only the physical layer (Layer 1), and perhaps Layer 2. All this means that whatever the network architecture may be, applications can treat is as a standard IP-based, packet-switched network supporting all the usual protocols such as TCP, UDP, and ARP through the usual interfaces such as Berkeley's sockets.

Table 5.2: Network Architecture

Layer	Protocols	
Layer 3	IP protocol suite	access control protocol
	Mesh routing protocol	
Layer 2	IEEE 802.11 (any variant that suffices) or IEEE 802.22;	
Layer 1	(optionally, any accepting IP frames and supporting jam resistance)	

The color-coded Table 5.2: Network Architecture demonstrates the proposed architecture from the Layer 3 downwards. It is obviously an early version, containing simplified protocol references; the network has not been a research priority.

§2. The Platform Architecture

Of all the requirements specified, the platform has been allocated the bulk of them.

Table 5.4: Platform Requirements

Numeric ID		Network	Ctrl Station	Platform
0.1	"Brasidas" is a research prototype	X	X	X
0.2	Adopt the use of COTS components and open standards	X	X	X
0.3	Funding and Support is very specific	X	X	X
0.4	All Autonomous Mode functions must depend only on installed capabilities			X
0.5	Target engagement happens only in Teleop Mode		X	X
0.6	The network must be based on the IP protocol	X	X	X
0.6.1	The network must support IGMP	X	X	X
0.9	The network must implement at least an access control protocol	X	X	X

Table 5.4: Platform Requirements

Numeric ID					Network	Ctrl Station	Platform
1.0				Move over Flat Terrain			X
2.0				Move at Speeds Comparable to a Human			X
	2.1			Minimum platform top speed should be 6 Km/hr			X
	2.2			Motion and velocity commands should be platform-agnostic		X	X
3.0				Be fully teleoperable		X	X
4.0				Possess optical and IR sensors			X
5.0				Operate autonomously			X
	5.1			Autonomous Mode must be the default operating mode			X
	5.2			Autonomous Mode must be suspended in Teleop Mode			X
	5.3			Navigate autonomously			X
		5.3.1		Perform SLAM and pose estimation			X
		5.3.2		SLAM must support dynamic map updates			X
		5.3.3		Follow path between two points on the map			X
			5.3.3.1	Plan path to target			X
			5.3.3.2	Generate motion commands			X
			5.3.3.3	Path planning must accept dynamic waypoint list updates			X
			5.3.3.4	Path planning must accept dynamic map updates			X
			5.3.3.5	The waypoint list must be wiped upon a map reload			X
7.0				Operate for at least 4 hrs at max power load			X
10.0				Connect to platform		X	X
	10.1			Establish initial connection		X	X
		10.1.1		Initialize network link		X	X
		10.1.2		Retrieve platform-specific parameters		X	X
		10.1.3		Retrieve additional platform-specific and payload capabilities		X	X
	10.2			The initial connection must follow a common protocol		X	X
	10.3			The functionality of the initial connection's common protocol must be implemented on all platforms regardless of payload			X
11.0				Stream data feeds			X
	11.1			Capture data feeds of onboard sensors			X
	11.2			Determine if there is a network request to stream data feeds			X
	11.3			Stream data feeds over the network			X
	11.4			Each stream should be manageable independently from others		X	X
	11.5			The NavCam stream must have a latency of 150 msec or less			X
12.0				Store area map			X
	12.1			Load or reload a map			X
	12.2			Retrieve map-related sensor data			X
	12.3			Process sensor data to produce map-specific information structures			X
	12.4			Update the map			X
	12.5			Map update rate must be optimal (TBD)			X
13.0				Track Location via GPS			X
14.0				Signal the platform		X	X
	14.4			The signaling mechanism must be part of the initial connection's Common Protocol			X

Table 5.4: Platform Requirements

Numeric ID		Network	Ctrl Station	Platform
14.5	The signaling mechanism must be request- and response-agnostic			X
16.0	Advertise platform over the network			X
16.1	Broadcast platform ID and status over the network			X
16.2	A platform must advertise at least once every 3 sec			X
16.3	A platform must not advertise more often than once every 0.6 sec			X
16.4	A platform must advertise when its location since its last advertisement has changed by at least 1 m			X
17.0	Disconnect from platform		X	X

Requirements R-1.0, R-2.0, R-4.0, and R-7.0 apply to the hardware, and can be ignored when designing the software architecture. Beyond that, it makes sense to separate functions that need to directly interact with the hardware, with those that don't. The aim is to abstract and parameterize the functionality as much as possible, so the same functionality can be used for another platform. One shouldn't forget that "Brasidas" is a research prototype, thus any product coming out of the project must be adaptable to any platform it may end up being used. Functions that do not need to directly interact with the hardware (such as SLAM or the common protocol), are prime candidates for this kind of abstraction. Functions that interface with the hardware must be rewritten for each platform they will be used on.

Naturally, the operating system will help abstract away much of the hardware interaction, but some demand for specialized code will remain.

Because the platform is allocated functions that perform tasks independent of one another (such as SLAM and platform advertising), it makes sense to consider multi-threaded or multi-process options. In this case, it is necessary to have some method of inter-thread or inter-process communication (IPC).

Communication among threads is usually trivial, as they run within the context of the same process, and thus have access to global variables and the stack, and, with proper argument passing to methods running within threads, the entirety of process variables

can be exchanged between threads. In the case of threads it is imperative that a thread synchronization primitive be applied where needed, to avoid variable value inconsistencies, as well as resource deadlocks. However, a single multi-threaded application is really just a bloated beast waiting to burst on some poor user's (or worse, developer's) face.

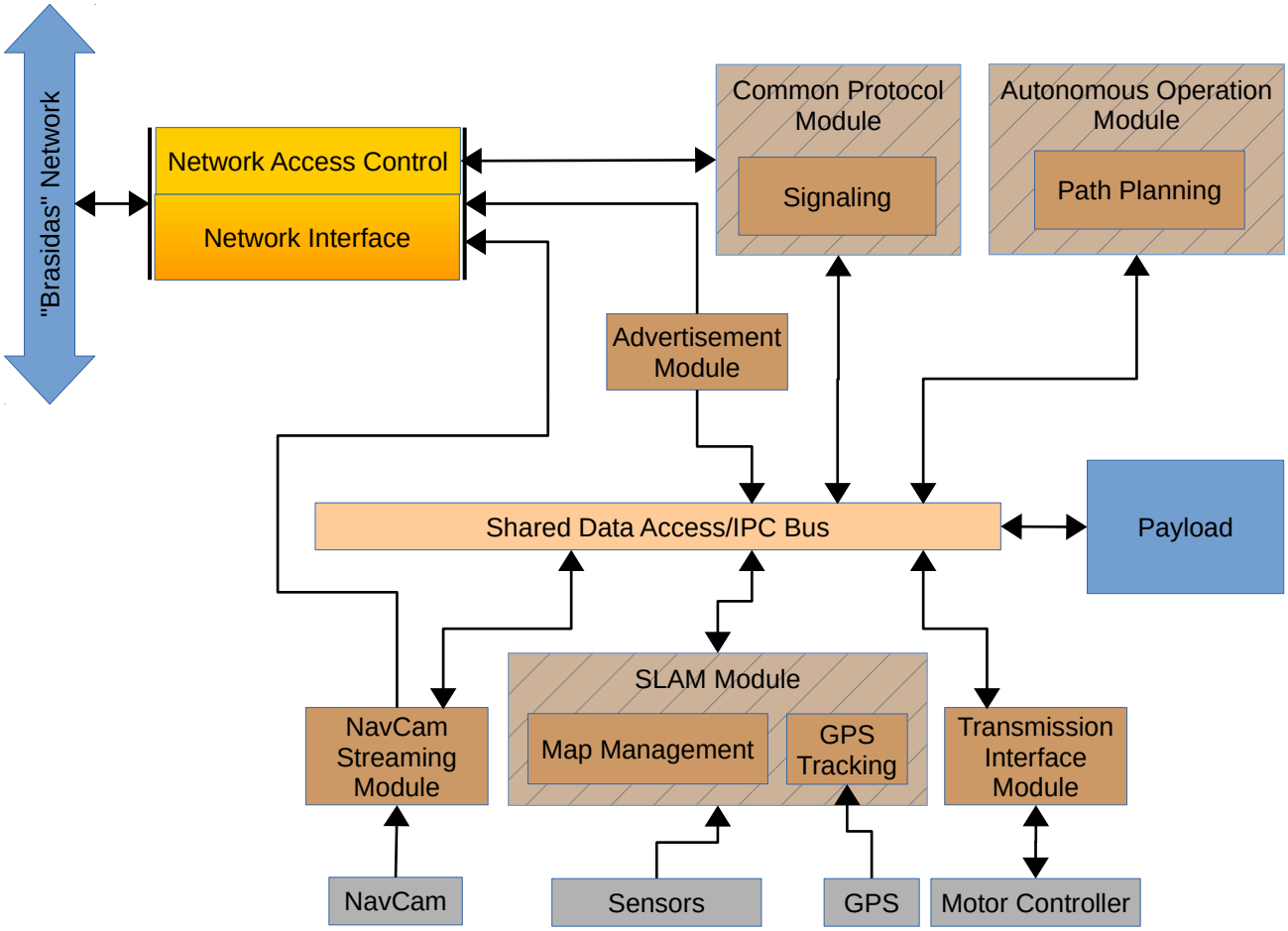

Figure 5.1: Carrier Vehicle Architecture

A multi-process solution is far more elegant, as each process's code can be kept small and lean, with little overhead. The downside is that this imposes a multitasking requirement on the hardware used, and context switching between processes is far more costly (in terms of CPU time and resources) than context switching between threads. Also, most processes are unable to directly communicate with one another due to process isolation (present on all modern multitasking OSes), and therefore the use of some IPC protocol is necessitated. Still, even given these shortcomings, the code simplicity comes on top, and multi-processing is the solution of choice when performance is not an issue.

The architecture depicted on Figure 5.1: Carrier Vehicle Architecture is designed with multi-processing in mind. It is color-coded; **solid blue** represents external software

components, **bright blue** represents visualization components, while **brown** represents background components, that interact only with other processes. Hardware interfaces are shown in **gray**, and **light brown** represents the IPC component. The Common Protocol, SLAM, and Autonomous Operation modules, whose internal components are shown, use a brown-based coloring variant.

The NavCam Streaming Module is also used as the basis of implementation of all other data streaming modules.

The reasoning of Figure 5.1: Carrier Vehicle Architecture is pretty straightforward. The Common Protocol Module is the only receiver of incoming communication; all requests and commands to the platform go through this module, using elements of the Common Protocol. This module also handles signaling, since its specification is part of the Common Protocol. All other vehicle operations are implemented as independent process modules. Certain modules, like Autonomous Operation and SLAM, also integrate already defined functionality in a convenient manner.

All onboard process modules communicate via an IPC bus, using some messaging protocol. This IPC protocol is implementation-specific; it does not form part of the system's requirements, since it is required only as part of the architecture proposed in Figure 5.1: Carrier Vehicle Architecture. Another architecture proposal could very well do away with IPC altogether, in which case there would be no point in imposing an implementation-specific requirement.

Figure 5.1: Carrier Vehicle Architecture does not present the whole platform architecture. The architecture of the payload must also be specified. Since there can be multiple payload types, there is no single fully-detailed architecture that describes them all, but it is possible, given the constraints and requirements specified for the payload, to present an abstract architecture to which all payload types should conform. This architecture is presented in Figure 5.2: The Payload Generic Architecture. Module names in italic indicate a generic type module, that may or may not be present in all payloads, as indicated by the respective multiplicity indicators ("0..n", "1..n").

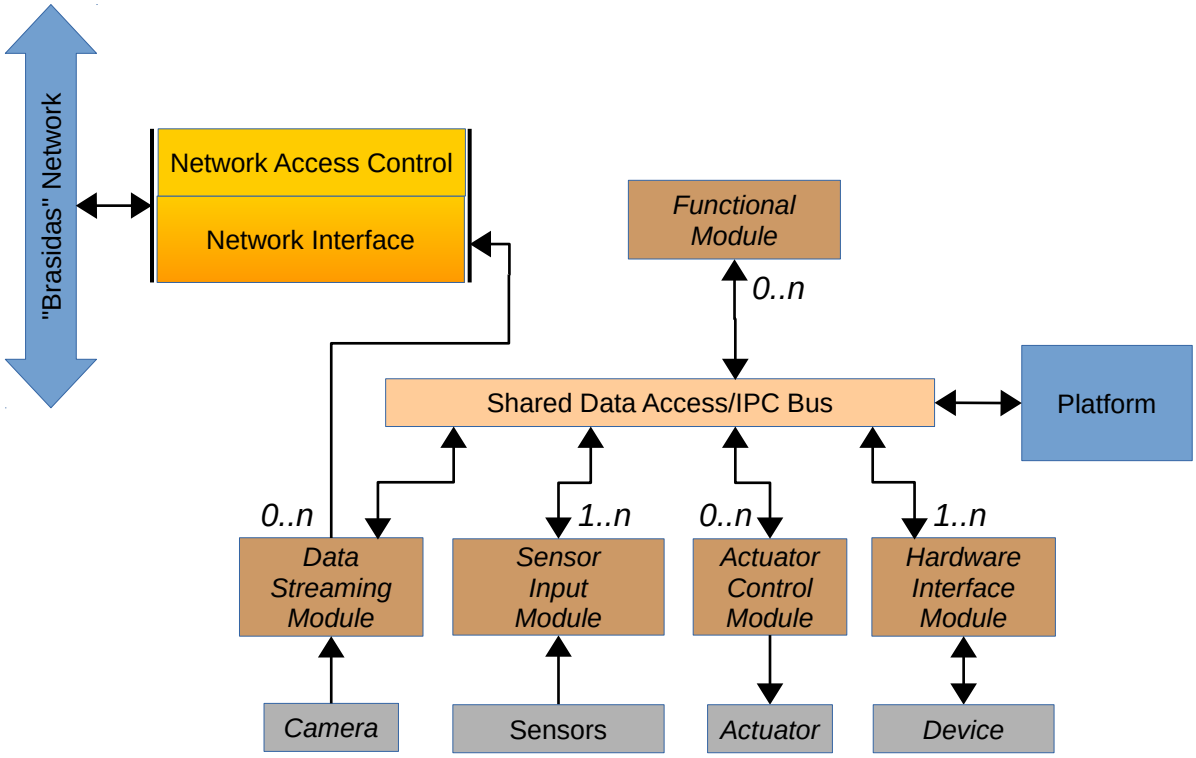

Figure 5.2: The Payload Generic Architecture

What is evident from Figure 5.2: The Payload Generic Architecture is that the common IPC bus is present, and is the only route of communication between the payload and the carrier vehicle. Also, the payload is afforded direct network access for streaming modules, in order to eliminate a potential mediator module that would otherwise have to be running on the carrier vehicle. Eliminating the mediator affords the data stream the minimum possible latency.

It is assumed that the payload includes at least a sensor and some hardware that the software needs to interface with. Otherwise, there's not much sense in needing to install an entire separate computer board, and not simply treating it as part of the carrier vehicle's System Core component. Given the distributed nature of the architecture, either the payload's processing component, or the carrier vehicle's System Core, need not be limited to a single processor board, and can instead easily be composed of multiple boards, in a form of mini-cluster. It's the IPC part that binds everything together seamlessly.

§3. The Control Station Architecture

The Control Station has been allocated the following requirements:

Table 5.3: Control Station Requirements

Numeric ID				Network	Ctrl Station	Platform
0.1			"Brasidas" is a research prototype	X	X	X
0.2			Adopt the use of COTS components and open standards	X	X	X
0.3			Funding and Support is very specific	X	X	X
0.5			Target engagement happens only in Teleop Mode		X	X
0.6			The network must be based on the IP protocol	X	X	X
	0.6.1		The network must support IGMP	X	X	X
0.9			The network must implement at least an access control protocol	X	X	X
	2.2		Motion and velocity commands should be platform-agnostic		X	X
3.0			Be fully teleoperable		X	X
8.0			Connect to the "Brasidas" network		X	
9.0			Discover platforms online		X	
	9.1		Add a node to the list when its status is received		X	
	9.2		Timestamp discovered nodes with time of reception		X	
	9.3		Remove nodes whose timestamp is 4 sec or more older than system time		X	
10.0			Connect to platform		X	X
	10.1		Establish initial connection		X	X
		10.1.1	Initialize network link		X	X
		10.1.2	Retrieve platform-specific parameters		X	X
		10.1.3	Retrieve additional platform-specific and payload capabilities		X	X
	10.2		The initial connection must follow a common protocol		X	X
	11.4		Each stream should be manageable independently from others		X	X
14.0			Signal the platform		X	X
	14.1		Send signal request		X	
	14.2		Receive acknowledgement		X	
	14.3		Receive response and return it to request initiator		X	
15.0			Display streamed data		X	
16.0			Disconnect from platform		X	X

To implement R-8.0 requires the proper hardware to begin with. From the software perspective, an appropriate device driver is needed. Usually, devices implement Layers 1 and 2 in firmware, and rely on the driver functionality to act as a bridge between the higher-Layer protocols, implemented in software, and the Layer 1 and 2 protocols implemented on the device.

Since the network is based on the IP protocol, the protocol stack specification is known (and is also an open standard). Any commercial product has all of this functional-

ity implemented in full, and the complementary higher-Layer protocols are also implemented for every operating system available.

R-8.0 and R-0.6 thus are grouped in the Network Interface component, as shown on Figure 5.3: The Control Station Architecture, which is closely coupled with the Network Access Control (R-0.9) component. The two modules are independent, to facilitate replacing one solution with another (e.g. changing access control protocols) while the rest of the system remains the same. However, network access uses both components in parallel, as indicated by the thick black lines delimiting the parallel combination of the two components.

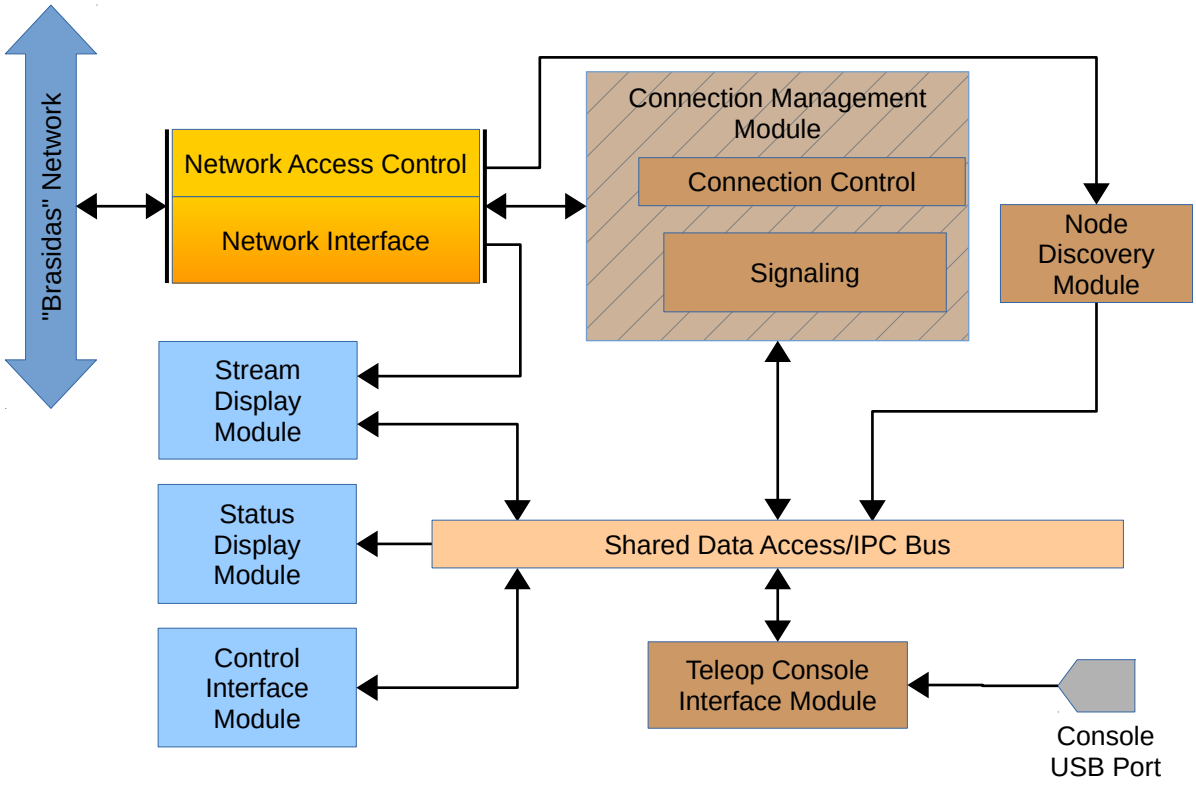

Figure 5.3: The Control Station Architecture

Figure 5.3: The Control Station Architecture is color-coded like Figure 5.1: Carrier Vehicle Architecture.

The proposed Control Station architecture also makes use of an IPC protocol, for the same reasons explained in the platform architecture section, above.

The Node Discovery Module is responsible for receiving all platform advertisements that come over the network. It encapsulates R-9.0 (Discover Platforms Online) and its descendant requirements.

The Connection Management Module is responsible for all communication with the robot. It handles connection and disconnection, sets up and tears down stream reception for the Stream Display Module, handles input received from the Teleoperation Console, and also implements signaling. It encapsulates R-8.0, R-10.0, R-14.0, and R-16.0 (with all lower-tier requirements of each).

The three visualization modules are responsible for letting the operator see and interact with the robot. R-15.0 is split between the Stream and Status Display modules, where the Stream Display Module renders on screen mostly the media streams (camera feeds, etc.), including "displaying" the incoming audio stream over the speakers, and the Status Display Module allows the operator to view non-media streams, such as telemetry and diagnostics updates. The Control Interface Module does not so much display incoming data, as providing a set of on-screen controls for sending commands to the robot, including manipulating the data streams (i.e. it encapsulates R-11.4).

The Teleop Console Interface Module is used to read in motion commands from a hardware control console, translating them to platform-agnostic motion commands, as per R-2.2 which it encapsulates. The console has special control hardware (joystick, buttons, MOVRD toggle) for optimal teleoperation of the robot. It is the hardware implementation of R-3.0, and is detailed in P. Katselis' thesis. As indicated on Figure 5.3: The Control Station Architecture, the Teleop Console module sends read in data to the Connection Management Module, which are then transmitted (in platform-agnostic format) to the platform via either the common protocol, signaling, or a specially set-up data stream.

CHAPTER 6 .
Product Development Phases

Though it may not be the effort to construct Star Trek™'s NCC-1701 "U.S.S. Enterprise" starship, "Brasidas" is still a rather ambitious project. To make matters worse, its design and development is undertaken by an inexperienced team of dabblers. Thus, to mitigate the enormous risks in time, effort, and monetary costs involved, the development must be meticulous.

The spiral development model is a good choice for developing a complex system with multiple risks present. However, behind it hides a rather insidious development threat: the model assumes those who adopt it do know from beforehand what they want to accomplish, but haven't figured yet out all the details.

In the case of "Brasidas", those who adopt the model **do not know a priori what they want to accomplish**, and that is a source of risk that even the spiral model does not cover. Thus, developing "Brasidas" is more than a matter of figuring out details; it entails building a knowledge base as well.

Consider also that the design is not only theoretical, but is accompanied by a prototype. To adopt Nielsen's terms [24], "Brasidas" is basically a horizontal prototype, as it attempts to encompass an entire system, not just some aspects of it (i.e. subsystems). Thus, the basic spiral model can only take development so far.

The above reasoning is what led the development team to built "Brasidas" as a research prototype (and hence imposed R-0.1). A research prototype can be a throwaway one [8], and therefore more mistakes are allowed. When the problem's scientific and engineering aspects have been studied sufficiently in depth, it might be deemed viable to then construct a second prototype, closer to a production model.

While "Brasidas" will not be the successor to the M1A1 Abrams tank, it is convenient to follow a similar approach; ToS incorporates that convenience as well. The paragraphs that follow attempt a description of the various development Phases, list the functionality that is intended to be integrated into each Phase, and give an estimate of the ex-

pected time required for a complete prototype, asuming full-time research from the same development team.

§1. Phase 1: Remotely Operated Vehicle

This Phase is expected to produce a prototype that can be remotely operated. The Mk-1 would implement all requirements and follow all constraints, except for those pertaining to autonomous operation and map management, namely R-0.4 (All Autonomous Mode Functions Must Depend Only on Installed Capabilities), R-5.0 (Operate Autonomously), and R-12.0 (Store Area Map). R-0.5 (Target Engagement Happens Only in Teleop Mode) is included here, since even though it may not be directly applicable to this Phase's functionality, the software design must consider it for when future Phases include autonomous options that call upon the software produced in this Phase.

The Mk-1 would be a R.O.V that can simply retrieve its location via GPS (since R-13.0 will be integrated). It could be teleoperated within known (pre-mapped) environments using the NavCam feed and perhaps a few additional simple sensors (such as sonars). It would feature a variety of remotely-operated payload options, such as a sensor cluster, small arm or an Infantry support weapon with IR targeting camera, an EOD turret with manipulator arm, perhaps a mine detector payload, etc.

Development of at least a token payload option is part of Phase 1. As a long-range, possibly armed R.O.V., "Brasidas" Mk-1 would be a solution ready for adoption and use by potential clients now, demanding minimal to no changes in the organizational structure and operational MOs of clients.

It is estimated that this Phase will require 1.5 – 2 years to produce a working prototype. As of May 2017, this Phase is complete, though a few software bugs may still remain to be solved.

§2. Phase 2: Recon R.O.V.

This Phase also concerns a R.O.V., except this time with additional functionality that

enable its use in unknown (unmapped) environments. Phase 2 includes all Phase 1 functionality, and also incorporates the following additional requirements: R-5.3.1 (Perform SLAM and Pose Estimation) and R-12.0 (Store Area Map).

The "Brasidas" Mk-2 will be fieldable in a reconnaissance role, to move into unknown areas and gather intelligence for advancing units. It could go into areas where a natural disaster (such as a flood, earthquake, or fire) makes the environment hazardous to human operators. Viewed from a more military perspective, it makes an excellent "first man in", luring out ambushes and triggering or detecting traps and minefields (exact capabilities will depend on installed payload options). If armed (and armored!), it can even engage enemy positions, providing heavy weapons support fire or simply forcing the enemy to reveal their precise positions (and thus be safely neutralized by accurate artillery or mortar fire).

SLAM and map management require additional sensors and processing capabilities, not demanded of Phase 1, so implementation of this Phase has impact on hardware design and integration as well. The full extend of this Phase's potential capabilities is still being researched, considered, and decided upon. A fully functional Phase 2 prototype is estimated to require an extra 0.5-1 years after Phase 1 is complete.

§3. Phase 3: Autonomous Navigating Robot

This Phase is where some autonomy finally begins to be integrated in the prototype functionality. This Phase includes all functionality of Phase 2 (and by extension, Phase 1), plus R-0.4 (All Autonomous Mode Functions Must Depend Only on Installed Capabilities) and the rest of R-5.0 (Operate Autonomously).

The major challenge of this Phase is path planning, since it must provide realistic, traversable paths in an outdoors environment. While the author is aware of several algorithms and methodologies out there that can solve the problem, only an attempted implementation will reveal what obstacles must really be overcome to make autonomous navigation feasible.

Payload options may or may not receive additional autonomous functionality during

this Phase; R-0.5 (Target Engagement Happens Only in Teleop Mode) still applies, but this Phase intends for all payload operations to still be controlled remotely (so, for example, the operator can traverse and elevate the gun turret, searching for potential targets, while at the same time the robot moves towards its next waypoint on its own).

A fully functional Phase 3 prototype is estimated to require an extra 2-3 years once Phase 2 is complete.

§4. Phase 4: Autonomous Target Tracking

This Phase is still considered far in the future. In concept, this Phase introduces mostly new payload functionality, enabling the platform to autonomously identify targets, and then track them, but not engage them.

"Identify" and "track" in this case refer to the specific payload used, and the same goes for "target". A mine detection payload would have mines as "targets", and would equate "identify" with spotting a mine, and "track" with putting its location on the map for others to avoid. An S.A.R. (Search And Rescue) payload would have survivors or humans in danger as "targets", and would equate "identify" with spotting such and individual and deciding whether he does indeed need help, and "track" with passing the location and status of the individual to the rescue teams. An armament payload would have enemies as "targets", and the terms "identify" and "track" here are pretty straightforward.

Development of this Phase primarily consists of evaluation and development or integration of suitable pattern recognition and DSP algorithms. Since all such algorithms are computationally intensive, development of this Phase may necessitate replacement or upgrade of the currently installed computational infrastructure. The choice of algorithms will obviously be tailored to the sensors installed on each payload variant. A potential solution is to improve the payload components, and leave the carrier vehicle as is.

Assuming Phase 3 is complete, development of a Phase 4 prototype is estimated to require an extra 2-4 years.

CHAPTER 7 .
Interlude I

At this point, the most important steps of defining a system's specifications are complete. The designers now know in great detail what the system needs to accomplish, and how well. However, a few more words must be said, lest one continues reading further under the impression that this has been a simple and straightforward process. Despite appearances, the readers should be assured that the case has, in fact, been quite the opposite.

As was stated earlier, the design team has adopted a modified spiral development model, and deviated from the strict DoD process by relying on scenarios and use cases to determine requirements that were not at first apparent. This revisiting of concepts, though not stated explicitly, has indeed happened many times during both the design phases and the implementation phases. In fact, several of the scenarios and use cases you read are the final versions of the system vision; for some, like the supervised autonomous patrol scenario, the original idea was vastly different. A different idea sends the design process down a different path, and produces different requirements. Sometimes, retracing the design steps and changing direction is easy, but when the process has moved too far, going back is hard, messy, and frustrating, and one is left with the feeling that this whole process was nothing more than a waste of time and effort. In such a case, typically one just scraps the whole process and starts over, as retracing is too costly in terms of the time required. In addition, the whole process gets too error-prone when the revision must be very extensive.

Other scenarios, pertaining to versions of the system vision far in the future of the development process, that have not been presented here for the sake of brevity, have been revisited and revised time and again to such an extend that noone in the design team remembers anymore what the original idea was. Revisions of a concept mean that requirements must be revised as well, thus many requirements also are presented in their final versions. Some were revised as a result of a change in the system vision, as mentioned

in the previous paragraph. However, others were dropped or introduced because the hardware required to implement a desired functionality was never acquired, or what was acquired had different specifications than expected, either better or worse. If the case was for the worse, it necessitated adapting to the reduced perfomance; if it were for the better, the temptation to take advantage of the new functionality was, naturally, irresistible (to what engineer is it not?).

Nevertheless, an effort has been attempted in this Part I, to present the requirements and proposed architectures of the envisioned system, to the extend these were shaped by the design team's experience acquired both during first designing the system, and subsequently trying to implement the various designs. The system proposed at the end of Part I differs from the system that currently sits in one of the ARTC's labs, which may represent the product of Phase 1, but is still not a finalized version of that product.

What is also not obvious is that not every requirement presented in this Part was conceived and specified at the beginning. Requirements were not just revised as the concept evolved and the design progressed; they were also introduced when the design progress reached a point where the next stage of the system vision became clear and concise, and the design team could actually get down and design this next advancement of the system, because concrete results could finally be produced. The main reason of this "in-medias-res" design and evaluation was the lack of experience and knowledge on the part of the design team. Knowing that a planar scanning laser can allow SLAM and path planning is one thing; knowing exactly what limitations and processing demands this solution has, is quite another. As the design team became more involved and acquainted ourselves with the minutiae of each prospective solution, it often became apparent that the solution was, after all, not as suitable as it had appeared at first. When that happened, it was time to go back to the drawing board and, well, rethink matters.

One last source of requirement revisions was the simple fact that as the implementation progressed, it became apparent that they didn't represent the system functionality in a realistic manner – in short, they represented design errors. Naturally, an error meant

that it was time to go back to the drawing board yet again.

One such example is the use of the `motion` application for video streaming. While initially seeming suitable, and allowed a functional streaming solution to be developed in but a day, `motion` proved to stream using TCP, and thus the video stream had very high latency. The troublesome behavior worsened when the connection quality was dropping, due to the increased number of packet retransmission requests, magnifying the problem of controlling the platform to a level that was simply unacceptable.

Several such examples will be listed in Part II, when the actual implementation will have been presented and the reader will have a more complete picture of the whole process.

Part II

Design and Implementation

CHAPTER 8 .
Architecture Implementation

To save on development time, and produce results quickly, the development team decided on using Python as the primary programming language. If for some parts of the implementation Python proved inadequate, it is always possible to fall back to C++, which interoperates very easily with Python and natively with Cython. Cython has not yet been considered on a serious basis, mostly because there is a Python package out there for virtually every need.

Python has proven very popular with robotics researchers and hobbyists [25], particularly since it is the de facto programming language for the Raspberry Pi single-board computer [26]. Its abundance of utility libraries, adequate execution speed, and low time required to write something that runs have practically displaced all other languages, except perhaps the ever-powerful C++. The fact that Python is written in C, and can interface with routines written in C very easily and very efficiently only means that Python will continue to rise in popularity. It therefore makes sense to base development of "Brasidas" on a language that will not become obsolete anytime soon.

§1. Common Architecture Elements

1.1. Operating System

On this matter, Linux was the only practical choice, since it is the only open-source OS with sufficient support and development to actually be usable. Of all Linux distributions, Debian and its derivatives (mainly Ubuntu) are the most popular and afford the greatest software availability. This choice was also restrained somewhat by the hardware specifications, as it had to be an OS that was supported by the Raspberry Pi computer boards that would be used for the platform and Control Station, as shown on Figure 8.1: Physical (Hardware) Architecture (taken from P. Katselis' thesis, and used with permission). The RPi Foundation provides Raspbian, a Debian-variant that suits the project's needs just fine. Raspbian has up-to-date repositories that include full driver support for

the hardware, including several – very convenient – packages that take advantage of the RPi's specific hardware capabilities. "Brasidas" initially ran the earlier Raspbian Wheezy distribution, it now runs the latest Raspbian Jessie.

1.2. The IPC Module

With the programming language and OS choices figured out, the most important piece of software to determine was the IPC module. Both the platform and Control Station software require it, so it made sense to use the same toolkit for both. In fact, this not only streamlined the code between the two, it also greatly simplified signaling. According to Figure 5.1: Carrier Vehicle Architecture and Figure 5.3: The Control Station Architecture, signaling is handled right after the network stack, and dumps most of its output into the IPC bus, so essentially one signaling module is talking directly to another. In addition to making the tranfer of signals easy, this permits direct transfer of data blocks, both structured and otherwise, by simply implementing a serialization/deserialization protocol alongside signaling. The transferred data would simply be grabbed off one IPC bus, be serialized, transported over the network, be deserialized, and injected directly into the IPC on the other side, available for process by any node that is interested.

As it turns out, there is just one such toolkit, and it's tailored for robotic applications; it is called R.O.S. (Robot Operating System - [27]). Which is a good thing, because having to write an IPC framework would constitute a thesis of its own. And ROS of course comes with full C++ and Python bindings. ROS is extremely popular among robotics researchers, and several books outlining its use have been published [28], [29]+[30].

The core philosophy of ROS is an IPC architecture relying on the concept of topics and messages. A topic is a channel through which data are transported in the form of messages. Each topic handles a specific message format, and messages can be user-defined. Each message is basically a data structure (similar to C's `struct`). A central server process (`roscore`) manages the topics. Multiple processes can register with `roscore` and gain access to one or more topics as either subscribers (receiving messages posted on the

topic), or publishers (putting messages in a topic). The important feature is that when a message is posted in a topic, all subscribers registered into it will receive a copy of the message, as soon as it is posted. This is achieved via callback functions. Each process that registers to receive messages on a topic specifies a function (part of the process code) to be called when a message arrives; the message is then passed to this function as an argument. A process registered with ROS is called a 'node'. A fully-detailed tutorial on publishing and subscribing to ROS topics using Python is available on the ROS site[1].

Messages can be all sizes and formats (the developer can define their own message format to suit their needs), and the whole message transfering process is done in memory, so it is as fast as it gets. The shared access nature of topics makes ROS an ideal choice for the IPC Bus module of both the platform and Control Station proposed architectures. It can even be used to transport continuous data (i.e. for streaming), however it should be pointed out that the topic transport mechanism uses TCP, and TCP can introduce delays in the transport even on localhost configurations (see e.g. [31]).

ROS features an entire ecology of robotics software packages, most of them by third parties. These packages are implementations of various robotics-related algorithms and architectures, interfaces to hardware devices, visualization tools, etc. For example, there are packages that act as interfaces to GPS devices, reading in the data from the GPS and posting it in appropriate topics, packages that implement SLAM or even a full-featured navigation stack (intended for the developer's sensor suite, though), robotic arm manipulation, etc.

In fact, ROS even works in a distributed configuration, where individual nodes can run on different machines connected to the same network, however there is a little catch: there can be only a single `roscore` instance running on the network. The distributed nodes feature would solve the signaling implementation across the entire network in an ideal manner, since signaling would reduce to a pre-defined set of topics shared among all nodes. However the network architecture rejects any form of centralized control on the basis of it being a "single point of failure", so this ROS functionality cannot be used over the entire network. But it can be of use locally on each node (platform or Control Sta-

1 http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)

tion), as will be explained in the respective implementation chapters. A different way for nodes to communicate over the network must then be employed.

Using ROS for interprocess communication, and implementing distinct functionality sets as separate Python modules also has the advantage of effectively bypassing Python's Global Interpreter Lock (GIL), which would otherwise render even a single-process, multi-threaded approach unattainable (the subprocess-based `multiprocessing` module does not permit easy access to shared resources like the `threading` module does).

§2. Hardware Specifications

Figure 8.1: Physical (Hardware) Architecture shows the system's hardware architecture; it is taken from Cpt. (AA) Katselis' thesis and used with permission. The system's modularity is obvious.

Here, a short set of hardware specifications will be provided, only to the extend that it relates to the developed software. These are as follows:

Table 8.1: System Specifications

	Carrier Vehicle
System Core:	Raspberry Pi 3 Model B
CPU:	1.2 GHz Quad-core ARMv8 Cortex A-53 (32/64 bits)
RAM:	1 GB LPDDR2 at 900 MHz
GPU:	300 MHz (3D part)/400 MHz (video part) Videocore-IV Supports GLES 2.0. MPEG-2 and VC-1 (with license), 1080p60 H.264/MPEG-4 AVC high-profile decoder and encoder.
Network:	RJ-45 port (10/100 Mbps 100BASE-TX auto-negotiation) 802.11n wireless Bluetooth 4.1
USB:	4×USB 2.0
GPIO:	40-pin header with I^2C, I^2S, UART, SPI, and GPIO interfaces.
NavCam:	Creative Livecam VFO470 (webcam)
Interface:	USB 2.0
Video Format:	640×480@30 fps, video encoded as either MJPEG or I420
GPS:	UBlox NEO-6M
Interface:	TTL UART
Sentence Format:	NMEA-0183
IMU:	MPU-6050
Interface:	I^2C

Table 8.1: System Specifications

Accelerometer Range:	±2/4/8/16 g
Gyro Range:	±250/500/1,000/2,000 degrees per second
Wireless Link:	Ubiquiti Bullet M2
RF Power:	28 dBm TX
LAN ports:	1×RJ-45 10/100 Mbps LAN port
Motor Controller:	Roboteq AX-3500
Interface:	RS-232 (9600 bps, 7E1)
Ethernet Switch:	3Com 8-port 10/100 Mbps

Control Station

Telemetry & Video Processor:	Raspberry Pi 2 Model B
CPU:	800 MHz Quad-core ARMv7 Cortex A-7 (32 bits)
RAM:	1 GB LPDDR2 at 900 MHz
GPU:	250 MHz Videocore-IV Supports GLES2.0. MPEG-2 and VC-1 (with license), 1080p30 H.264/MPEG-4 AVC high-profile decoder and encoder.
Network:	RJ-45 port (10/100 Mbps 100BASE-TX full-duplex, auto-negotiation)
USB:	4×USB 2.0
GPIO:	40-pin header with I^2C, I^2S, UART, SPI, and GPIO interfaces.
Human-Computer Interface:	Touch-screen Display
Diagonal:	10.1"
Resolution:	1366×768
Interface:	HDMI (video) + USB (touch sensor)
Input:	Capacitive touch sensor
Teleop Console:	Logitech Gamepad F310
Interface:	USB
Wireless Link:	Ubiquiti Bullet M2
RF Power:	28 dBm TX
LAN ports:	1×RJ-45 10/100 Mbps LAN port

As has been mentioned repeatedly, the payload does not have a fixed hardware configuration.

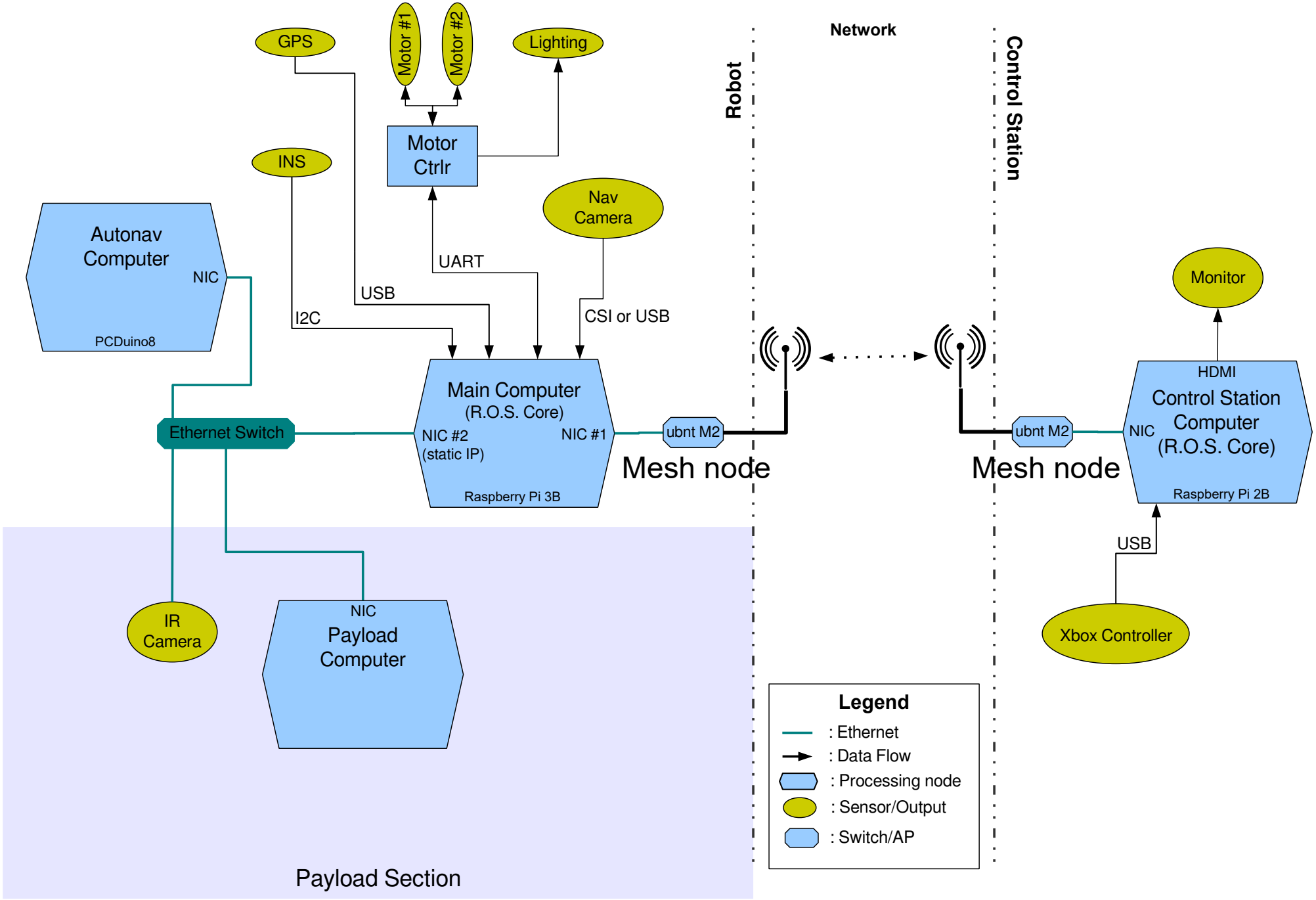

Figure 8.1: Physical (Hardware) Architecture

CHAPTER 9 .
The Network Infrastructure

As was stated in Chapter 5.1: The Network Architecture, the network is very basic; there are two nodes in a trivial mesh configuration.

Following constraint R-0.3, the only enterprise-grade link hardware that was made available are a pair of Ubiquiti Bullet M2s. These are civilian, enterprise-grade WiFi transceivers, so again there is little choice on the matter. They are currently configured in mesh mode, and both routers have been flashed with the AREDN™ mesh-enabled OpenWRT distribution [32].

The network in this configuration uses static addresses in the 10.x.x.x range for the mesh nodes (i.e. the mesh wireless interfaces of the routers), and relies on dynamic addressing (DHCP server is enabled) over a predetermined, different subnet, for the devices connected locally to each node. The way the onboard systems are currently linked, every subsystem connected to the Communications Grid (i.e. the onboard Ethernet) is directly visible to the Control Station or any other computer joining the network. This is not what is intended, but for the time being, it permits the system to also access the Internet (using a gateway that connects to the network's AP), allowing system updates of the RPi and software package downloads to be implemented as and when needed. Obviously, when the system will be assembled as a production prototype, this arrangement will most likely have to change, although this may depend on the exact network configuration used.

To generalize the discussion a bit and refer to other solutions examined, the real deal about 802.11 is that unless one adopts a sub-GHz implementation, it really is basically an obstacle-free LOS communications option. Ubiquiti's NanoStation M9 is one rather popular option for the 900 MHz band. But the 900 MHz band is not so ideal either, despite such signals seemingly having a longer range than the 2.4 GHz band. The problem is the 900 MHz band (902-928 MHz) is ham radio territory, and as such, the RF interference is worse than even in the 2.4 GHz WiFi band.

Promising upcoming options are 802.22 [33] and 802.11af [34], which take advan-

tage of "white spaces" in the TV channel frequencies (54-790 MHz), using the bandwidth leftover after the transition from analog TV to digital broadcasting. Both standards promise data rates of the order of tens of Mbps, over distances of more than 1 Km, and in the case of 802.22 several tens of Km (LOS) – naturally, these advertised ranges imply directional antennas. However, given the transmission properties of these frequencies, even NLOS links would function well over the 1-Km limit of "Brasidas"'s R-6.0. No commercial products implementing these protocols have been released as of yet (April 2017), and in addition, according to the initial draft standard specifications, mesh (ad-hoc) mode is not to be supported.

Even sub-GHz solutions remain LOS-dependent, although as the ferquencies get lower, obstacle tolerance increases. However, without delving into specifics, a NLOS communications link typically requires rather low frequencies to efficiently bypass obstacles (a typical limit is less than 160 MHz, depending on range and nature of obstacles), and the data rates that can be achieved at such low-frequency bands are also low. A custom, from-scratch design *might* be able to squeeze a couple Mbps at a 50-MHz band, but that's about as good as it could get, and it violates R-0.2 (use COTS components). In addition, to achieve such a data rate at this band would require higher-order modulations, which are typically less resistance to interference and jamming.

Note that at frequencies above approximately 30 MHz, the link cannot extend over the horizon (as there is no ionospheric refraction or ground wave propagation), but can still go through many kinds of obstacles. The downside to using lower frequencies is that the antenna gets progressively bigger.

Any WiFi-based (IEEE 802.11) solution can be made to satisfy all the requirements of Table 5.1: Network Requirements, as long as there is no inclusion of a requirement for jam resistance. Consumer-grade 802.11 solutions do not satisfy the range requirement, however there are enterprise-grade products with more powerful transmitters that can easily reach the distance of R-6.0 (e.g. Ubiquiti's line of AirMAX stations and TP-Link's line of MAXtream-based stations). In fact, there are some very nice 802.11 solutions out

there that far exceed the specifications of Table 5.1: Network Requirements.

The typical problem encountered with enterprise-grade equipment is that it is designed for infrastructure installations, and thus most such models do not support mesh topologies. Furthermore, the consumer need that spurred development of these devices is for long-range *static* point-to-point links (to provide Internet to hard-to-reach locations), thus the range they advertise is often measured using directional antennas. "Brasidas" is a mobile system, so omnidirectional antennas are required, and one can quickly understand how max range diminishes when comparing a good 9-dBi omni antenna to a directional 24-dBi one; 15 dBi of difference translates into an approximate 1/6th of advertised maximum range, and that's still under a LOS assumption. Of course, when certain products have an advertised maximum range of 15 Km+, they can satisfy R-0.6 sufficiently even at 1/6th of that range.

One solution to the mesh mode availability, applying to most Ubiquiti and certain TP-Link products, is to flash them with custom firmware. Specifically, a mesh-enabled version of OpenWRT [35]. This is one option that the development team intends to explore further in the future, as each of the candidate products has a steep cost of the order of 60+ €, not including additional peripherals, such as antennas, cables, etc. Mesh-enabled OpenWRT distributions include HSMM-MESH™[1] ([36]) and AREDN™[1] ([32]), both products of ARRL member groups. Right now the AREDN™ distribution is used, however due to FCC regulations, no access control is enabled in the distribution (i.e. no WPA), so further research is needed to determine how to implement mesh using either the base OpenWRT image for the routers, or alternatively another distribution (like the related DD-WRT [37]).

Perhaps the most suitable, easier-to-acquire fully military-standards-complying solution in this matter, given the company's proximity (to Hellas in general, not just to the Military Academy of Athens), is Intracom Defense Electronics' WiWAN system. But this assessment is based only on the limited information provided on the company website's product page[2], which simply mentions "high data rate" without giving any numerical de-

1 Why on earth would someone trademark an open-source product anyway?
2 https://www.intracomdefense.com/post/410

tails; however it does mention that it is intended to interconnect brigade command echelons with battalion and company ones, so given the expected distance between these echelons during typical operations, it is safe to assume its effective range is over 1 Km. Otherwise, similar military-grade solutions are available from a multitude of defense industries (e.g. MeshDynamics[1]). Such solutions come at the expected price of course, each approaching a typical household's yearly revenue.

Back to the current implementation, WiFi is a thoroughly-tested, well-understood technology, and works fine for the time being. The RF power of the M2s (28 dBm) satisfies R-6.0 more than sufficiently; in fact, over open ground, the link theoretically has a range of several Km (LOS), even when using omnis. Since this configuration serves the project for the time being, no additional options have been tested. Thus, while the issue of Control Station – platform communication is settled for now, several issues still remain unresolved.

According to the specifications (R-16.0), the platform needs to broadcast its advertisement packets, and the Control Station needs to be able to receive those advertisement packets. However, both computers sit behind their corresponding routers, and routers break broadcast domains. After several (failed) efforts to configure the firewalls to allow broadcast packets through, in the end the design team settled for using `socat` on the OpenWRT routers. Specifically, a `socat` instance is run at startup (via an entry in `/etc/rc.local`). For the Control Station router, `socat` forwards broadcast packets received on port 21000 at the external network interface to the LAN, while for the platform router, `socat` forwards broadcast packets originating on port 21000 of the internal LAN to the external network. The details are provided in ANNEX B at the end of this work.

1 http://meshdynamics.com

CHAPTER 10 .
The Platform Configuration

The chassis was provided by the ARTC (remember constraint R-0.3), and included a motor controller. An RS-232 interface permits communication with the controller. Thus, from a functional and software point of view, the system needs to adhere to the controller's RS-232 command protocol.

A demonstration payload was developed, consisting of a turret with a paintgun and an AXIS Q1910-E thermal IP camera. The camera was again provided by the ARTC. The turret is rotated by servos controlled via an Arduino, and communicates over Bluetooth, in direct violation of the current payload proposed architecture, but this demonstration payload is still under development.

Each module box shown on Figure 5.1: Carrier Vehicle Architecture is implemented in code as a separate Python module. Virtually all of the modules are registered as ROS nodes as well, and use ROS to communicate among themselves. The software stack on the platform is started by a single command, using ROS's `roslaunch` process, which reads in an XML-formatted file and launches one by one all applications listed therein (be they ROS nodes or not!). As part of the same procedure, a configuration file is also loaded automatically. This configuration file contains platform-specific parameters, such as the node name of the UART port to which the motor controller is connected, or the network port used to stream the NavCam video feed. This permits the code to be portable to other platforms, simply by changing the appropriate parameters. The process stack launch procedure at the moment requires the operator to login manually via `ssh` and issue the `roslaunch` command, but efforts are underway to implement it as a linux service, enabling the software to start automatically at system boot.

§1. Signaling and the Common Protocol

Signaling has until now been specified completely abstractly, and has been presented as functionality that is separate from the Common Protocol. In general, this assumption

holds; however, in the current Phase's implementation the two functional sets are intertwined, although this is not cast in stone, and could be revised in a future Phase (highly unlikely though).

From a network developer's point of view, the Common Protocol is nothing more than an application receiving requests and sending responses over the network. The first request received is usually from a remote Control Station attempting to establish a connection, and subsequent requests involve exchange of the platform's specific parameters. Thus, the Common Protocol module is nothing more than a network server, listening for incoming (request) connections on a specific port, then servicing each request over a separate connection (and port). All that remains is to define the request and response formats.

However, the issue can be viewed from a different perspective. The Common Protocol can be considered as two objects of the same application that exchange information by each calling (some of) the other's methods. Extending this view to include the network element results in the RPC (Remote Procedure Call) paradigm. Implementing the Common Protocol as a set of RPC interfaces will greatly speed development, as there will be no need to write tedious transport-layer code, and instead approach the solution from a high-level point of view, encapsulating functionality parts in self-contained function calls.

```
import Pyro4

@Pyro4.expose
class Exposable(object):
    def method1(self, param1):
        return "Received {0}.".format(param1)

daemon = Pyro4.Daemon()                 # make a Pyro daemon
uri = daemon.register(Exposable)        # register the Exposable class as a Pyro object

print("Ready. Object uri =", uri)       # print the uri assigned to the daemon
daemon.requestLoop()
```

Listing 1: Pyro4 sample daemon code

Python has Pyro4 (see [38]), a pure-Python package that can wrap a Python object and expose its interface to the network. Although Pyro4 follows the RPC paradigm, its overall approach is in fact more similar to Java's object-oriented RMI, and is perfectly

suitable for use in modern, object-oriented applications.

The only downside to Pyro4, which is really only a minor inconvenience, is that objects exposed to the network need to be registered with the Pyro nameserver, in order for another computer to be able to discover them and call upon their methods. However, if the remote process has somehow acquired the uri of an exposed object, it can refer to it and call its methods without the need for a nameserver. Thus, the Pyro4 uri of a platform's Common Protocol class is transmitted as part of the advertisement packet.

Listing 1: Pyro4 sample daemon code (taken from Pyro4's online documentation) shows how quick and simple it is to expose the interface of a class to the network. Pyro4's `Daemon` class wraps the exposable class and does all the work. Note that it is possible to expose a class, or an already instantiated object; in the first case, Pyro will create an instance of the class when a remote request arrives. Each approach (expose class or object instance) has its merits and flaws.

```
import Pyro4

expo = Pyro4.Proxy(uri)          # get a Pyro proxy to the Exposable object
expo.method1("test")             # call method normally
```

Listing 2: Pyro4 sample proxy code

Listing 2: Pyro4 sample proxy code shows how simply and elegantly a remote client can use the exposed object of Listing 1. The approach and implementation follows the Proxy design pattern. The client does not even need to include the file specifying the exposable object's method signature.

Pyro also includes additional options that determine, among others, how multiple remote requests are handled. Possible options include delegating all requests to a single instance of the exposed class or creating multiple instances, and processing one request at a time or using a thread pool to concurrently access the exposed methods.

Since Pyro uses a very convenient URI string, the advertisement module needs to do nothing more but broadcast the platform's URI (and a few extra pieces of information) to the network using UDP, while a Control Station can just receive these broadcasts and built its own list of platforms online. This will also permit keeping the platform list up-

dated.

This rather mundane initial setup might seem somewhat complicated, and was initially intended to permit sending just the MOVRD signal coming from the Command Station, via ROS to the other onboard modules. However, after some deliberation, it inspired the design team to use it to good advantage in implementing signaling, which until that point was considered under a different context. The signaling process will be explained below.

1.1. The Virtual Functional Remote Interface (VFRI)

The Pyro4 library handles all the server/connection/data (de)serialization issues, leaving the developer free to focus on the problem at hand: in this case, the Common Protocol implementation itself, which consists of a ROS node that exposes a Virtual Functional Remote Interface (VFRI) Python object to the network. This object is detailed below.

The `vfri` includes all those methods needed to pass or receive platform-specific parameters with a connected Control Station. In addition, the `vfri` is subscribed to the onboard ROS IPC Bus, and thus can relay information received over it to the rest of the onboard modules.

Listing 3: The "Brasidas" VFRI Docstring shows the Python docstring for the `vfri`. In it can be seen the various methods currently implemented as part of the Common Protocol. Of these, the most important method is `issue_system_command`. This method alone implements the signaling mechanism. Its parameters and usage are explained below, in the section titled Commands and Command Codes.

The other `vfri` methods are pretty self-explanatory. `set_manual_override` is used to toggle the MOVRD of the robot; this method returns a boolean, indicating whether MOVRD was changed successfully (True) or not (False). This sort of feedback is required on the Control Station to enable or disable certain status panel indicators and control options. The three `get_*` methods are used to retrieve platform-specific parame-

ters, as is demanded in the Common Protocol requirements; the dictionary returned by `get_nav_camera_parameters`, for example, contains four entries, with keys 'width', 'height', 'cam_ip', and 'cam_port'. 'width' and 'height' specify the dimensions (resolution) of the camera feed, and are needed by the Control Station to properly resize the window of the Stream Display Module (cf. Figure 5.3: The Control Station Architecture), while 'cam_ip' and 'cam_port' contain the IP and port respectively of the NavCam video stream sent over the network, so the Control Station knows where to expect to receive the video of the platform it has connected to.

```
@pyro.expose
class VFRI(object):
  """ Vrasidas Function Remote Interface

  Sole class that acts as a 'public' interface to Vrasidas
  (follows the 'Facade' design pattern).
  The class features methods which expose certain functions
  of the Vrasidas ROS-based software architecture so they can
  be accessed remotely.
  Pyro4 (PYthon Remote Objects v.4.0) serves as the remote
  access bridge (follows the 'Proxy' design pattern).
  Supports the following methods:

  issue_system_command(cmd_group, cmd_id, cmd_params) : sends a generic system command
                                                        (signaling)
  set_manual_override(engaged)              : engages or disengages Manual Override
                                              (MOVRD)
  get_nav_camera_parameters()               : returns a dictionary with the navCamera's
                                              parameters
  get_telemetry_report()                    : returns a dictionary containing current
                                              telemetry values.
  get_velocity_connection_info()            : returns a tuple (host, port), indicating
                                              where to send velocity commands
  set_lights(state)                         : turns the main lights on (state=True) or off
                                              (state=False)
  set_pos_lights(state)                     : turns the positional lights on (state=True)
                                              or off (state=False)

  """
```

Listing 3: The "Brasidas" VFRI Docstring

The implementation is still in a transitional state, and several of the methods shown in Listing 3, will be removed eventually, in particular `get_telemetry_report`, `set_lights`, and `set_pos_lights`, as they violate the new Common Protocol specification. These methods where part of earlier code versions, when signaling was still considered only theoretically and in a piecemeal fashion. Another method, `set_velocity` (now removed and replaced) was called by the Control Station several times per second to transmit velocity commands, and was very inefficient and laggy, since Pyro4's proxy

mechanism relies on TCP. `get_telemetry_report` is also called a few times per second (fewer than `set_velocity`) to retrieve the current telemetry values. This 'pull' approach will be replaced by a stream-based 'push' one on the platform's side.

Currently, as per the data stream specification, all continuous data transfers must be implemented as data streams. Thus, an attempt is underway to re-implement the functionality of `get_telemetry_report` using data stream mechanics, which will be as simple as sending UDP datagrams to predetermined address/port pairs. `set_velocity` is rather easy, as the data required to be sent every time is small and already determined, and has already been implemented as a UDP stream. `get_telemetry_report` is more difficult, as the size of the telemetry dictionary is as of yet not finalized, and likely to eventually grow rather large. The tasks performed by `set_lights` and `set_pos_lights` have already been included in signaling, as are listed below, and the methods are obsolete and not really called anymore. They will be removed along with the others when the data streams for velocity and telemetry are complete.

1.2. Commands and Command Codes

The single `issue_system_command` method of the `vfri` object, in combination with a ROS topic, can handle all the signaling required. This method will simply accept a signal structure, and inject it into a specific ROS topic, to which every onboard module subscribes and monitors. Each module can then examine the injected signal, and decide if it wants to act on it. This approach closely mimics the MPI concept (see [39]), but implements only the functionality desired in a lean, simple fashion, and avoids having to import an additional dependency which would include a lot more code that would be used.

Signaling thus is implemented in two layers. The top layer, which is the *Signal Specification Layer*, defines the signals that can be sent to the platform. Signals are called commands, and are organized in groups. Each group has a (numeric) Group ID, and each command within the group has a Command ID; Command IDs are unique within the same group, but not necessarily so across groups. Each command can possibly have pa-

rameters or arguments (its 'payload'), and all payload options are passed along as a single string of comma-separated '`<key>=<value>`' pairs. This is basically a trivial implementation of the Command design pattern. Part of the current (not yet complete) list of supported system commands is shown on Table 10.1: "Brasidas" Command Codes.

With respect to the entries on the table, Group 10 contains the primary system control and configuration commands, while Group 20 contains telemetry control commands. An additional 11 groups (100-110) are reserved for payload-specific commands, and will be defined as part of each payload's specification.

Table 10.1: "Brasidas" Command Codes

Group ID	Command ID	Description	Payload Options
10	0	Manual Override (MOVRD)	"True" or "False"
10	1	Enable (start) all PiGstPLBase streams	Host address
10	2	Disable (stop) all PiGstPLBase streams	Ignored
10	3	Pause all PiGstPLBase streams, resume with 'start' command(10.1)	Ignored
10	5	Change framerate of specified PiGstPLBase video stream.	"<stream_name (string)> = <new_framerate (int)>"
10	6	Turn main lights on or off.	"ON" or "OFF"
10	7	Turn positional lights on or off.	"ON" or "OFF"
20	0	Enable diagnostics report generation	"True" or "False"
20	4	Enable monitor of Primary (motor) battery voltage	"True" or "False"
100-110	ALL	Payload-specific commands (11 groups are provided to allow to group together in distinct sets, payload commands corresponding to different components of the payload).	Varies

PiGstPLBase is a base class that manages a Gstreamer pipeline; more on the use of Gstreamer below. As is obvious from the table, commands 10.6 and 10.7 implement the functionality of `vfri`'s `set_lights` and `set_pos_lights` methods. Internally, both these methods, as well as `set_manual_override`, just call `issue_system_command` with appropriately formatted arguments.

The important feature is that the signaling mechanism does not need to know the innerworkings of each command; as long as the payload is delivered, a recipient module that knows how to handle the command will know how to parse the payload string.

The lower signaling layer, the *Signal Transport Layer*, takes advantage of the Pyro4/ROS combination. It consists of the aforementioned `vfri` method and a predetermined ROS topic. The `vfri.issue_system_command` method receives the three

command parameters (Group ID, Command ID, Payload) from the remote Command Station, then formats them into a custom ROS message called `SysCmd`, written by the design team for this purpose, and publishes the message into a (platform-local) ROS topic.

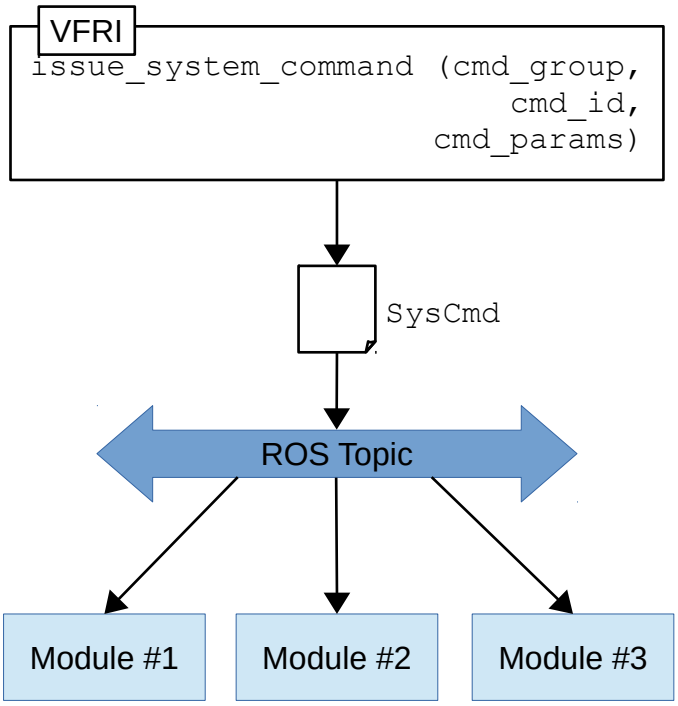

Figure 10.1: The Signaling Procedure

The topic is named '`/System_Commands`', and all modules running on the platform are subscribed to it. Thus, the signal gets propagated quickly and seamlessly across all platform modules, and the module responsible for handling it can do so. Under this approach, even the MOVRD is implemented as a signal command. This process is depicted on Figure 10.1: The Signaling Procedure.

Despite the signaling specification requiring that a response be possible to be returned to the signal originator, the current implementation adopts a middle ground. Those commands for which a response is necessary (such as whether the MOVRD was successfully engaged or not), are implemented as separate methods of the `vfri` (e.g. the `set_manual_override` method). The rest of the command options, are handled through the single, generic `issue_system_command` method. The effect of these commands can be determined by examining the appropriate fields in the telemetry information that is periodically streamed back to the Control Station. This current signaling implementation is considered very elegant and handy, and is unlikely to be revised in future Phases; effort will be made instead to keep the number of commands that demand an immediate response – hence requiring a separate `vfri` method to be implemented for each – to a minimum.

In effect, each module running on the platform follows in a limited fashion the Finite State Machine pattern; when a command is received via the '`/System_Commands`'

topic, the module may change its behavior to match that imposed by the command. However, for many modules, this state change is rather trivial, and usually is implemented in a straightforward manner, without complicated State class implementations and the like. For example, when the Control Station sends a 10.6."ON" command (turn on the main lights), the module responsible simply sends an appropriate request over the RS-232 to the motor controller (in hardware, the lights are controlled by the controller as well)

An identical structure is implemented on the Control Station, but there it is used to disseminate notifications, as well as commands. This will be explained in the Control Station chapter that follows.

§2. The Transmission Interface Module

In order to move the robot, the motor controller needs to receive velocity commands. Since the motor controller is connected to the RPi over a serial connection, care must be taken to control access to the connection, since it cannot by definition handle multiple simultaneous requests. The module that handles all this is the Virtual Motor Controller Interface, or `vmci`. The use of the term 'motor controller' instead of 'transmission' dates back to the early development phases, when the more abstract hardware architecture was not yet established. Back then (early fall of 2015), the major focus of development was how to communicate motor commands to the controller, since the thing has a rather archaic serial protocol.

2.1. The Motor Controller RS-232 Command Protocol

The motor controller accepts motor velocity commands in different modes, depending on whether wheel encoders (or any other form of wheel speed measurement) are connected to it or not. For teleoperation, encoder information is superfluous (the operator can adequately judge the velocity from the navigational camera's feed and the rest of the telemetry), so we adopt the mixed-mode, open-loop speed control mode. In this mode, velocity commands are given as a linear component along the x-axis (forward/re-

verse) and an angular component around z-axis (turn left/right)[1], with the magnitude indicated by integer values ranging from 0x00 to 0x7F (in hex). For the same velocity value, a different command prefix denotes the forward or reverse direction. The controller can also accept velocity commands in non-differential (absolute) format, with one linear velocity component for the wheels on each vehicle side.

The motor controller's RS-232 protocol supports additional functionality in the form of various kinds of telemetry queries and control commands, including but not limited to, battery voltage monitoring, motor current monitoring, control of up to eight RC-signal-conformant servos, and an auxiliary power output. Some of these functions can and will be integrated into the system as implementation proceeds from one Phase to the next.

All commands (except a few specific exceptions) are prefixed with either a combined '!' and alphanumeric one-letter symbol that identifies the command, or a or a '^', followed immediately by the command parameters. Any numeric values (such as e.g. a speed value) must be given in hex, and all numeric values are limited to one-byte range (0-255), which is always given as a two-character hex value (e.g. 5 = 05, 111 = 6F, etc.). the responses are likewise scaled to the same range. Of the two kinds of commands, the '!X' ones are run-time commands, while the '^' commands are used to change the controller's configuration parameters stored in its flash memory, such as the motor control mode or ampere limit to the motors. All commands must be followed by a `carriage return` character (`\r`) in order to be accepted by the controller.

In mixed-mode speed control, speed commands in particular use the '!A' prefix for the x-axis forward component and '!a' for the reverse. Likewise, the z-axis command uses the '!B' prefix to steer left, and the '!b' to steer right. The x-axis sign was specified in the controller's manual, but the relation between the z-axis prefix and the direction of rotation was not specified in the manual, and the design team had to figure it out using established scientific methodology – that is, through trial and (mostly) through error.

The controller's RS-232 interface is factory-configured at 9600 bps, 7 data bits, even

1 For coordinate conventions in ROS see http://www.ros.org/reps/rep-0103.html

parity, 1 stop bit (9600 7E1 for short). These are fixed settings that cannot be changed. When the motor controller receives a command from the serial port, it echoes back the command string, then sends back a response, indicating whether the command was received and executed successfully or not.

2.2. The Virtual Motor Controller Interface

To establish effective communication with the motor controller, in an abstract manner according to the architecture's hardware abstraction requirements, the `vmci` is written as a ROS node that manages the serial port and performs no other task. In true object-oriented form, the `vmci`'s main code is contained in a Python class, with module-level code handling the initialization tasks.

To communicate with the serial port, the `vmci` makes use of the `pyserial` library (`python-serial` package), which simplifies serial access in Python to a single line of code, as opposed to the 50+ lines (and `termio` structures) required in C/C++. Since on linux the serial port is line-buffered, each command must be followed by a `newline` character (`\n`) in order to be immediately transmitted. This is in addition to the `carriage return` required by the controller's protocol.

The `vmci`'s main class is also subscribed to several ROS topics. Of these, '`/System_Commands`' is used to receive signaling; the `vmci` currently responds only to the 10.0 (MOVRD) and 20.4 (enable/disable primary battery voltage monitor) commands.

ROS conventions demand that motor commands be published in a topic named '`cmd_vel`'. Note that this need not be a top-level topic (no '/' prefix). Thus, and since R-5.1 (Autonomous Mode Must be the Default Operating Mode) applies, the `vmci` subscribes initially to this topic, so it can receive speed commands from the Autonomous Navigation module. However, when a MOVRD engage is received via signaling, the `vmci` switches its subscription over to the '`Teleop_Vel_Cmds`' topic, which contains speed commands received from the Control Station in Teleop Mode. Likewise, when MOVRD is disengaged, the `vmci` switches back to the '`cmd_vel`' topic.

Since speed commands can arrive on the topic at any rate, and the callback used to

handle each topic message runs on a separate thread, it is possible that the `vmci` might end up trying to write to the serial port before a previous write operation is finished. Thus, a lock (via the `threading` library) is used to prevent simultaneous writes. This synchronization primitive is well-known and will not be detailed here further.

The `vmci` also has the capability to query the controller for certain telemetry values. The controller can report on the battery voltage, the temperature of its MOSFET-based dual H-bridges, and the state of several analog and digital inputs available on the controller board. When these values are retrieved by the `vmci`, they can be injected into an appropriate ROS topic, so along with others they can be compiled by another node into a telemetry report that can be streamed back to the Control Station. This functionality has not yet been implemented, but it is part of Phase I development, so it will be worked upon in the immediate future.

As a module that directly interfaces with the hardware, the `vmci` would be one of the modules that would have to be re-written or replaced if "Brasidas" would ever be installed on a different platform with a different motor controller (or Transmission subsystem technology in general).

§3. The NavCam Streaming Interface

The Virtual NavCam Streaming Interface (VNCSI) module is one of the modules that implement the new data streaming architecture. The streaming of the NavCam video feed in particular, is a common element on all platforms regardless of payload. Video streaming, due to its high bandwidth and CPU requirements (for encoding/decoding), as well as analog information nature, is the most complex form of data streaming. Other data streams, such as the telemetry report stream or the velocity commands stream (the latter is incoming to the platform from the Control Station) may require exact bit delivery to be meaningful, but have much lower bandwidth and processing requirements.

3.1. The `motion`-based Implementation

The previous version of this module was implemented on top of `motion`, a free open-source streaming server program intended for surveillance applications [40]. `motion` is an excellent application, and can be configured for all kinds of environments and client needs, but its use was a quick and dirty solution, for while it was implemented in but a day, it had significant shortcomings.

For one, the `motion` process ran separate from its controlling node, and has no notion of ROS, so this essentially meant that a process integral to "Brasidas" ran without being directly connected to the IPC Bus. The way to control `motion` was through a RESTful HTTP api, so the controlling node basically used the `requests` python package [41] to communicate commands to `motion`, while itself being connected to ROS.

The worst problem with `motion` was that it streamed video using software encoding as MJPEG. This method takes up significant bandwidth and CPU, and with unacceptable latency. As an example, during the system testing with `motion`, a 640×480@30fps video required almost 5 Mbps of bandwidth, a 65%+ utilization of one of the RPi 2B's cores (before it was uprgaded to a RPi 3B), and the video latency never managed to get below 1 sec. To achieve even these results, the video quality had to be reduced to the bare minimum. The high latency was because `motion` assumes a static, wired camera setup, and thus streams video using TCP. As time passed and errors required packet retransmission, the stream lagged more and more behind reality, and when bandwidth improved, one would get a sudden quick burst of frames – similar to the rubber-banding effect one experiences when playing an MMORPG with bad latency. Clearly, if that was the price to pay for a single stream, transmitting more than one stream by using for example an additional camera on the payload, was prohibitive.

3.2. The Gstreamer Era

The current video streaming solution is based on Gstreamer 1.0, a free, open-source toolkit for media streaming [42], and takes advantage of its Python bindings, available via the GObject Introspection repository. A set of stream handling Python classes is written

(all based on the PiGstPLBase abstract class mentioned on Table 10.1: "Brasidas" Command Codes), that each provides a native Python interface to the respective GStreamer pipeline. These classes are shown in Listing 4: The PiGst* Class Descriptions, which is taken from the header comments of the source Python module that defines them.

```
# Classes included herein are:
#
# - PiGstPLBase     : Base class for all the rest gstreamer pipelines defined in here.
#                     Implements common behavior.
# - PiGstUDPThru    : Implements a UDP "pass-through" pipeline, i.e. one described
#                     by 'udpsrc ! udpsink'. This is used to forward internal video
#                     feeds outside Brasidas, or to forward incoming feeds to internal
#                     processors.
# - PiGstVideoOut   : implements an RTP-over-UDP video stream FROM onboard Brasidas
#                     to an external client (i.e. Control Station). The video is
#                     encoded in H.264, utilizing the available openMAX extensions
#                     to take advantage of the hardware-accelerated H.264 encoder
#                     available in the Raspberry's VideoCore-IV GPU. Obviously this
#                     class only really works on boards that can support the specific
#                     openMAX extensions (i.e. omxh264enc).
# - PiGstAudioOut   : implements an RTP-over-UDP audio stream FROM onboard Brasidas
#                     to an external client (i.e. Control Station). The audio uses
#                     the Opus codec (free, open-source).
# - PiGstAudioIn    : implements an RTP-over-UDP video stream TO Brasidas (will be
#                     played back through the onboard audio out interface) FROM an
#                     external client (i.e. voice feedback from the Control Station).
#                     The audio uses the Opus codec (free, open-source).
#
# This file is used by the Video Server Node to control each registered camera.
# It is not meant to be run directly as an executable.
```

Listing 4: The PiGst Class Descriptions*

The base class, PiGstPLBase, defines the common functionality. It basically holds the Gst.Pipeline object used to implement the streaming pipeline, and defines the following methods:

- `start()`,
- `pause()`,
- `stop()`,
- `get_state()`.

These methods provide all the needed functionality for now. In a future Phase, it is unlikely the structure will be modified, but it may be expanded to include additional common functionality. Note that none of these classes is ROS-aware; they are simply convenience interfaces (wrappers) to Gstreamer pipelines. But they are used by ROS-aware nodes, such as the `vncsi`, to implement video streaming in a structured, elegant fashion.

As can be seen in Listing 4: The PiGst* Class Descriptions, the PiGstVideoOut class encodes video in H.264 format, using hardware acceleration that is available on the RPi. In fact, the capabilities of the RPi's VideoCore-IV GPU are quire amazing, as it is rated at an astonishing 24 Gflops, more than half the processing capability of a Core-i7! Using the OpenMAX extensions is easy for Raspbian Jessie and the latest Wheezy, as they are thankfully available on the repository (package `gstreamer1.0-omx`) – Older Wheezy requires building from source, which is a hassle[1]. Of course, one needs to install all Gstreamer required packages, as well as the `gir1.2-gstreamer-1.0` package, which contains the Gstreamer Python bindings (introspection data).

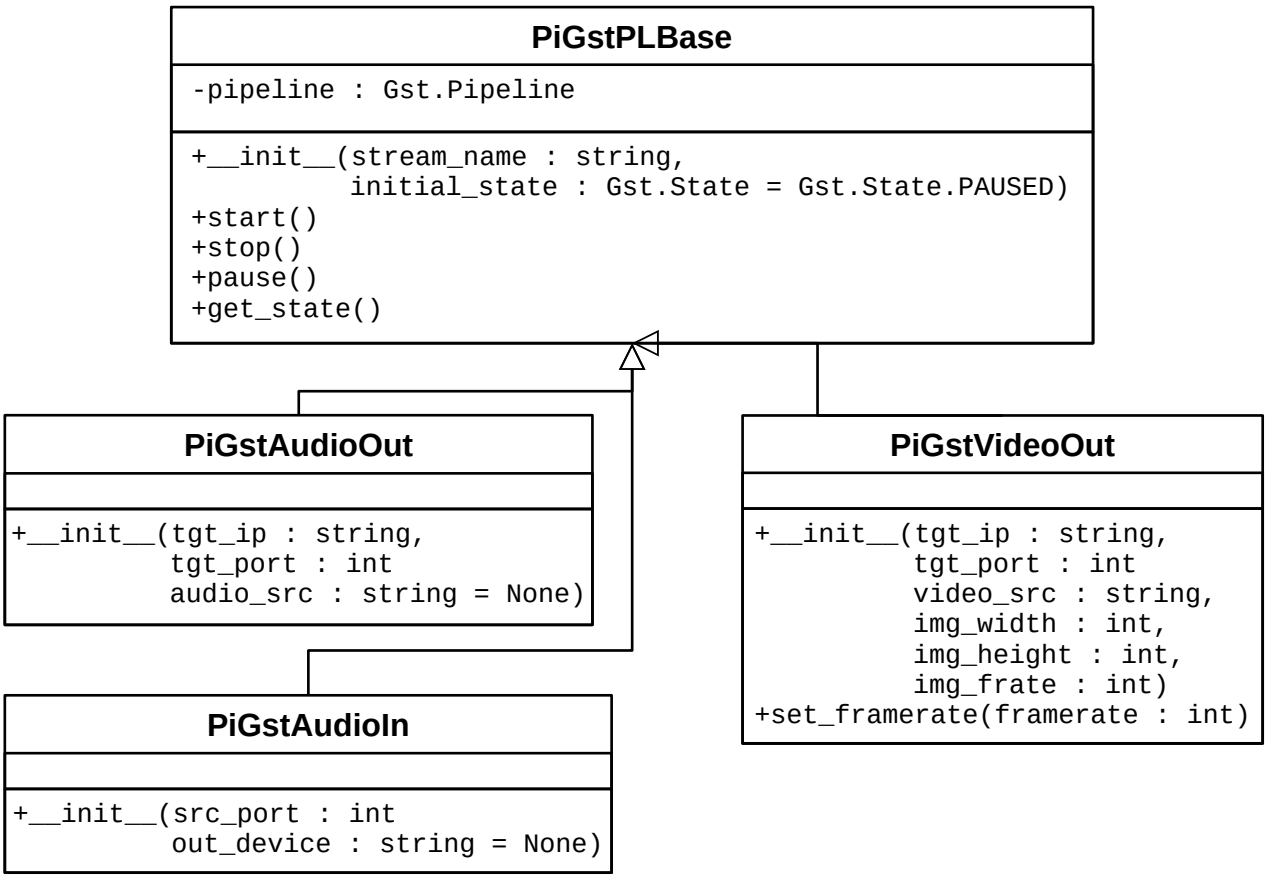

Figure 10.2: The PiGst Class Hierarchy*

The `vncsi` node uses the PiGstVideoOut class to transmit the NavCam's video feed. It is also intended to use PiGstAudioIn and PiGstAudioOut to implement the bidirectional audio link alongside the NavCam stream; this latter functionality is still in development. The NavCam stream is sent to the connected Control Station's address, which it

1 See ANNEX A for some instructions on where to start doing it.

receives via signaling (payload of command 10.1).

Figure 10.2: The PiGst* Class Hierarchy shows the relation between the streaming classes. PiGstPLBase is the base class, that includes an empty pipeline and defines the basic functionality. The `__init__` method is Python's class constructor. All classes' constructors also call the class's parent constructor before performing any other initialization (in other words, PiGstVideoOut's `__init__`, for example, also calls PiGstPLBase's `__init__`). It should be noted that for all classes of Figure 10.2: The PiGst* Class Hierarchy, the constructor arguments are also saved into additional "private" member variables, not shown explicitly on the diagram. The term is in quotes here, since Python does not implement any kind of strict access control, and all members of a class are publically accessible. The use of the term (and the '-' symbol in the class diagrams) indicates the architecture-derived intention, not its implementation.

Some constructor arguments also have default values. For the PiGstAudio* classes, the recording (src) and playback (out) system devices have a default value of `None`, and if this is not changed, Gstreamer will use the default system device for each operation.

The PiGstVideoOut class is configurable, so the network port, as well as other feed parameters, such as the camera source device, and video resolution, are passed to it as constructor parameters, making the class usable in other projects as well. The only restriction is that the video source must be from a Video for Linux-compatible device (the class uses the `v4l2src` pipeline element internally). The rest of the PiGst* classes are also configurable to a similar degree.

The fun part of using Gstreamer+OpenMAX is the performance. The streaming pipeline uses RTP over UDP to deliver the resulting H.264-encoded video, which at the same 640×480@30fps setup results in a 1.6-Mbps bandwith utilization, a less than 20% CPU utilization (on one core only), and a received video latency at the Control Station of less than 150 msec – as demanded by R-11.5. In fact, the CPU utilization is probably mostly due to memory copying of the camera frames to and from the GPU memory area during encoding. One catch is that the memory allocated to the GPU must be increased

to at least 128 MB, otherwise the H.264 encoder crashes. The RPi shares its memory between the CPU and GPU, and the setting that controls the memory allocated to the GPU is the `gpu_mem` parameter found in `/boot/config.txt`. Alternatively, one can use '`sudo raspi-config`' (without the quotes!) from an `ssh` console to change the setting. Both methods accomplish the same thing.

It should be noted that the H.264 encoder available on the RPi's GPU is a separate SIP (Semiconductor Intellectual Property i.e. chip section) from the main graphics pipeline. Thus, while the Videocore-IV GPU is encoding video, it can also service any graphics operations, or as intended in our case, generic GPU computations.

The only downside to using a Gstreamer pipeline, is that unless one sets it up to be receivable by `vlc` or a browser, the bare RTP-over-UDP stream can be opened only by another Gstreamer pipeline. This is not a problem in the case of "Brasidas", since the Control Station does indeed use Gstreamer as well, but it limits somewhat accessibility to the streamed video. Configuring the pipeline to make the stream viewable in `vlc` or a browser will also significantly increase the video latency.

To conserve bandwidth when not needed, in both the `motion` and Gstreamer cases the NavCam's framerate is reduced to 5 fps when MOVRD is not engaged. Simply monitoring the robot feed works fine at 5 fps. When the operator engages MOVRD, however, the framerate is switched to 30 fps, to permit smooth control.

Additional feeds can be streamed by using separate ROS-aware modules and making use of the appropriate classes from Listing 4: The PiGst* Class Descriptions. The Gstreamer solution is a recent one, so there was not yet the opportunity to expand upon it. Data streams will probably be implented by wrapping direct use of sockets and UDP into a higher-level, PiGst*-type class to facilitate code uniformity, although there is a possibility that one can take advantage of Gstreamer to stream custom data as well. The feasibility of this is still under investigation, however using a UDP socket in Python generally is much simpler than setting up even the simplest Gstreamer pipeline, so a custom solution is more likely in this case.

§4. The Advertisement Module

This functionality is currently implemented to transmit UDP packets at a network broadcast address (X.X.X.255). According to R-16.0 (Advertise Platform Over the Network), the minimum information that needs to be transmitted is the platform's ID and status. R-16.0 likewise specifies exactly the rate it needs to be transmitted at. The advertisement uses UDP port 20000 (rather arbitrary choice, but who's gonna object anyway?).

The platform's status is a 32-bit integer, accessed as a set of flags. A flag can be the typical 1-bit toggle (such as whether MOVRD is engaged or not, or whether the positional lights are on or off), or it might need to be a set of bits enough to differentiate between a set of options. For example, 2 bits could be used to specify a coarse battery condition, with possible values of "FULL" (`11`), "3/4" (`10`), "1/4" (`01`) or "DANGER - LOW" (`00`). If in the future these 32 bits of flags are exhausted, an additional 32-bit integer can be added to the status, or the status extended to a 64-bit integer, with minimal work. Efficiency hints at matching the integer size to the CPU's word, and the RPi 3's is 64bit. The current Raspbian Jessie comes only with a 32-bit kernel, however, so until that changes, a 32-bit CPU and OS will be assumed.

Of course, given that the typical ID will be a string of around 15-20 characters, sending 24 or so bytes using a UDP packet is a waste of bandwidth, as a UDP packet has an 8-byte header, increased by the at least 20 bytes of the IP header, for a total of 28 bytes of header and 24 bytes of payload, not including the link layer's framing bits.

After Phase I is complete and the development moves into Phase II, the advertised information will also include the robot's SLAM-derived coordinates. The coordinate system that will be used is yet to be determined. Given that the scale of applications for "Brasidas" is of the order of 1 Km or so, and that coordinates need no better accuracy than 1 m, this should not become much of an issue. Right now, the coordinates provided by the GPS are transmitted instead.

In any event, two coordinates need to be specified, perhaps a third if elevation is to be considered. It is assumed that in whatever form these are given, a 32-bit word for each

will be more than sufficient, so a Phase II advertisement packet would have about 36 bytes of payload (and 28 bytes of header).

All numeric values (status, coordinates) are transmitted as Base64-encoded strings. This adds a bit to the overhead, but allows transmissions to treat the data as a big string with fields separated by commas, rather than having to assume fixed-width byte fields.

Even adding 36 more bytes of status data, yields an IP packet that is 100 bytes long. Transmitting this at the maximum rate of R-16.3 (1 packet every 0.6 sec) requires only ~1400 bps bandwidth. This is still a very minimal network load, so much that if the knowledge of robot ID, status, and coordinates alone is enough, a separate, low-bandwidth, long-range network can be used to transmit those, such as the Xbee-based solution proposed by Cpt (Inf) Botonis Dimitrios and Cpt (Sig) Lefteris Takis in their semester project for the Robotic Systems course (spring 2015, under Prof. Mavridis Nikolaos), the same course which gave birth to "Brasidas". Since the advertisement module will run as an independent ROS node, it will be capable of using any hardware interface available and not restrict itself to network interfaces. In all likelihood, however, the implementation will not diverge from using a single radio (digital link), and advertisement packets will be routed over the IP-based network.

A more accurate bandwidth estimate should take into consideration the link layer overhead induced (error correction and coding), but even a ratio of 1:5 (transfer speed : link rate) as is typical in 802.11 (see [43]), is acceptable.

§5. The SLAM Module

This module is as of the time of this writing (Apr 2017) still in its infancy. Granted, its functionality is part of Phase II, while "Brasidas" still struggles to finish Phase I, but some notes can be provided.

SLAM is a CPU-intensive task, and every solution (i.e. algorithm design and implementation) must be tailored to the sensor data available; the observations matrix in particular must be structured differently when using a planar laser scanner than when using a stereoscopic camera and point cloud data. Now most implementations out there rely on

planar laser scanners, and it should be obvious that such mapping hardware may be appropriate for indoors applications, but it is whoefully inadequate for outdoors environments, with sloped ground, uneven surfaces, and random-circumference obstacles such as rocks and tree trunks. Thus, the design team is still in the process of attempting to determine what sort of sensor is appropriate and adequate. Of course, R-0.3 (Funding and Support is Very Specific) still applies. Once this is determined, then the proper SLAM implementation can be selected, or coded if desired. OpenSLAM.org [44] contains several algorithms to choose from before delving into state updates and filtering, and some of those are also ported to ROS.

One part of the SLAM module that is implemented is the GPS tracking component. The module is actually taken ready from the ROS repository of packages, and all it does is read in the NMEA-0183-formatted sentences from the GPS sensor, extract coordinate data, and post these in a ROS topic. No sweat.

Another functionality taken in part from the ROS package ecology is the IMU module. The IMU module is actually two separate ROS nodes. One is an in-house (i.e. written by the design team) IMU data acquisition node, which uses the class `MPU6050` given in [45], and along with a few ROS parameters and Python goodness, publishes the raw IMU data (accelerations and angular velocities) into the ROS topic '`/imu/data_raw`'. The data are in turn consumed by the IMU filter node, which integrates the instantaneous readings and publishing the robot's orientation (pose) into topic 'imu/data'. The filter node is available as an official ROS package[1], and uses a sensor fusion filter based on the algorithm described in [46].

§6. The Autonomous Operation Module

Much like the SLAM module, the Autonomous Operation (AutoOp) module has not been implemented. It is expected that this will contain CPU-intensive code as well, so it is planned that it will be run on a separate system Core computer, an additional one to

1 http://wiki.ros.org/imu_filter_madgwick

the RPi currently installed in the System Core. A tentative initial research has uncovered a ready-made ROS package for autonomous navigation. This will be examined and tested, in the hope that perhaps the AutoOp module can be based on that solution and save some code writing time.

Autonomous robot operation is still in its infancy, and most progress and ideas are purely theoretic. The DARPA Challenge to produce a driverless vehicle may have resulted in several viable prototypes and solutions [47], but these are mostly autonomously *navigating* robots, not autonomously operating in general.

There is still no standard among protocols and theories regarding autonomous operation. ROS includes a task-executive based package called `actionlib`[1], and other action-based approaches do exist, such as the military-compliant, NIST-sanctioned 4D/RCS [48]. When Phase IV development begins, the then-current status and options will have to be re-examined before a proper implementation can be chosen.

Autonomous operation is still debated among the design team as to whether such a capability is really desired of a robot like "Brasidas". An independent capability to track targets is definitely desired, and is intended to be included, but more complicated processes may be superfluous and never actually be used.

The design team's intent is to have Autonomous Operation permit the robot to prioritize a list of operator-provided tasks, and identify certain operator-defined targets (such as cars, doorways and windows, pedestrians, and so on). This will still demand significant onboard processing power, but will provide specific needed services, with as little possibility of error as possible.

1 http://wiki.ros.org/actionlib

CHAPTER 11 .
The Control Station Configuration

Currently, the Control Station exists only in software; its hardware is not yet complete. The software stack runs fine on a Core-i3-based laptop, and is expected to run just as well on a Raspberry Pi 2B or later, that is intended to be used as the Control Station processor.

The software stack is initialized in the same manner as the platform's software stack, using `roslaunch` to load configuration files and start nodes with a single command. The big difference between the platform code and the Control Station code, is that the Control Station features a GUI for human-computer interaction. The GUI layout is not finalized and is still under development; more on the matter in the Display Modules section later in this chapter.

§1. The Connection Management Module

This is the most important module of the Control Station, as it essentially routes network connectivity. Currently, it implements a set of services offered to the other nodes of the Control Station, that manage aspects of the connection to a remote platform.

The module functions in one of two states: Connected (to a remote platform) or Disconnected, with the module being by default in the Disconnected state at startup.

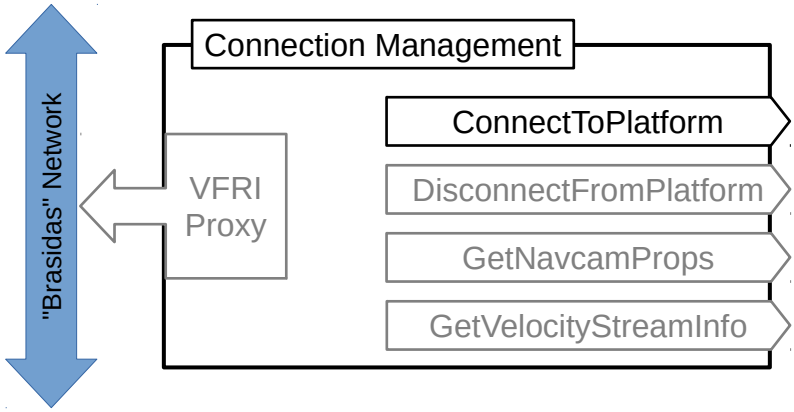

Figure 11.1: Connection Management module (Disconnected state)

Figure 11.1: Connection Management module (Disconnected state) displays the services in the Disconnected state. Here, there is no VFRI Proxy instantiated, since no remote platform is specified to connect to. Only the ConnectToPlatfrom ROS service is enabled. When the operator selects a platform from the displayed list of discovered platforms, the generated UI event calls this ConnectToPlatform service, which if successful, causes the module to change state to the Connected one shown on Figure 11.2: Connection Management module (Connected state).

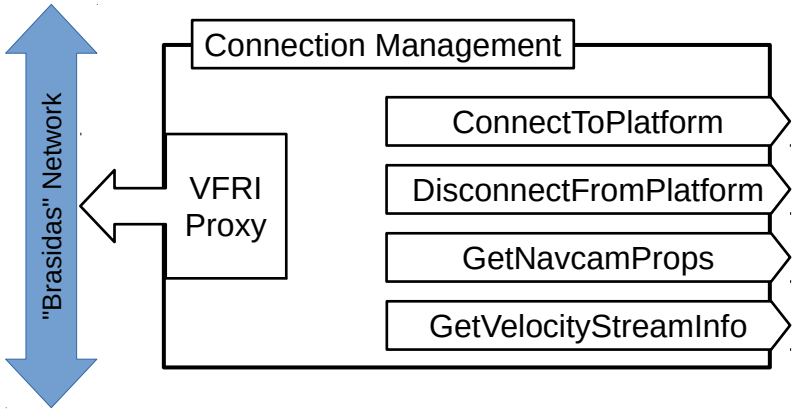

Figure 11.2: Connection Management module (Connected state)

When a connection to a remote platform's VFRI has been established, all the services offered by the module become enabled. The other modules of the Control Station use these services to receive information about specific platform parameters, for example, the GetNavcamProps service is called by the Stream Display module when a connection is establish, to retrieve the parameters (video dimensions, network port of incoming stream), so it can properly receive and render the NavCam stream from the platform.

The Connection Management module uses information supplied as argument to the ConnectToPlatform service call, to create a Pyro4 Proxy to the `vfri` of the platform selected by the operator. It also monitors the '`/CtrlStation_Commands`' ROS topic for any system commands (non-notifications) posted by other Control Station modules, that necessitate a call to the `vfri.issue_system_command` method, to forward the command to the platform.

Finally, the DisconnectFromPlatform service tears down the connection, releases the reference to the remote VFRI proxy, and disables all other services except Connect-

ToPlatform. Right now, it is required to disconnect from a platform before connecting to another, however when the current code revision is complete, the system will be able to switch connections on the fly, without having to disconnect first. This will permit faster control of multiple platforms by a single Control Station, e.g. to quickly issue a new way-point to another robot operating autonomously, then go back to the one currently being teleoperated.

Currently, the Connection Management module also pulls periodically telemetry data from the platform, and posts back the returned result into the '`/CtrlStation_Commands`' topic, using appropriate Group 90 commands (notifications, see below). However, this is a temporary solution, as it goes against R-11.0 (Stream Data Feeds); a revision is already underway to implement telemetry as a data stream (i.e. data will be "pushed" by the platform), that will be handled by a separate code module.

1.1. The Control Station Signaling

Signaling is handled a bit differently on the Control Station. While the overall software architecture mirrors that of the platform's, the mission of the Control Station software is fundamentally different from that of the platform's software stack.

The platform software works in an active manner, taking in sensor readings, executing Control Station requests, streaming media, etc. Inter-module signaling on the platform aims to propagate commands and effect status changes. The Control Station software, on the other hand, works reactively to respond to operator commands. Inter-module signaling on the Control Station intends to propagate notifications about data received by one module that necessitate a change in the visual interface presented, or data displayed by another module.

Thus, signaling on the Control Station consists mostly of *notifications*. The various modules of the Control Station keep tabs about the currently connected platform in the form of a set of various state variables, such as the platform's ID, the port where the platform's NavCam stream is received, whether the platform's MOVRD is engaged or not, whether the lights have been turned on or not, etc. All these variables remain valid in

the current state (their values may change, but their raison d'être does not). They are the equivalent of state variables in a thermodynamic system (the set of state variables and their relations is the equivalent of the state function of a thermodynamic system).

A notification informs all modules of a change in the collection of state variables. This may be a change in the value of a variable, or the addition or removal of a variable from the set of state variables. Thus, notifications are analogous to process functions in thermodynamics.

Notifications are propagated through the '`/CtrlStation_Commands`' ROS topic, and have command Group ID and Command IDs, like other signaling commands. Notifications are assigned the command Group ID 90, and are published ***only*** on the Control Station. Note that other commands can also be used on the Control Station, and indeed are, notifications (i.e. Group ID 90 commands) are simply restricted to the Control Station only. Most of the additional non-notification commands are forwarded to the platform, although Command Group 91 is reserved for (the future possibility) of Control-Station-only commands.

§2. The Node Discovery Module

This module is a simple UDP server at its core, listening on the network for platform advertisement packets. It is build on top of python's SocketServer module (specifically, it uses the `ThreadedUDPServer` class as base).

When running, the node maintains a dictionary of all robots currently discovered. If a new packet from an already discovered robot arrives, the node simply updates the dictionary entry with the new values. If a new robot appears, it is added to the dictionary.

Along with the data included in the advertisement packet (robot VFRI uri, IP address, coordinates, and status word), the module stores in each dictionary entry the (Control Station local) time of the last advertisement packet received from that robot. A separate thread runs every few seconds (10 in the current implementation), and cleans up the list of stale entries, removing robots for which the last advertisement packet is more than

10 sec old. The requirements (R-9.3) specify 4 sec, but 10 sec are used at the moment until the code is refined and stable.

Whenever a new packet is received, or the cleanup thread completes, the updated platform list is published in the '`/node_list`' topic.

§3. The Teleop Console Interface Module

Much like a reverse version of the `vmci` running on the platform, this module monitors the teleoperation console and forwards any button presses or joystick moves to the currently teleoperated robot. The teleoperation console is connected to the Control Station over USB, but it's really just a wired (USB) XBox game controller. A third-party ROS node, `joy_node` (from the `joy` package) is used to read in the controller's state. The Teleop Console Interface module simply picks up the messages published by `joy_node` and converts them to velocity commands for the remote platform.

3.1. Sending Velocity Commands

Upon receiving a notification in the '`/CtrlStation_Commands`' topic that the Control Station has been connected to a platform, the module calls the GetVelocityStreamInfo service (part of the Connection Manaement module) to retrieve the details needed to create the velocity commands stream (in particular, the platform's IP address and port where it receives velocity command packets). Velovity values are converted from the 16-bit signed integer values read in from the controller, to platform-agnostic 0.0-1.0 range `floats`.

In particular, the MOVRD enable/disable is routed over signaling, since it is imperative that its reception (and execution) be acknowledged. At the same time, the velocity commands are sent over a separate data stream. Such a stream (i.e. basically UDP destination host/port pair…) is initialized upon connection, and utilized only when MOVRD is engaged. In fact, since a UDP-based solution is actually connectionless, "connecting" and "disconnecting" is really a software flag toggle. By simply changing the stream's parameters (host and port), the velocity commands stream can be directed at a different platform

on the fly.

Right now, velocity command packets are sent at a rate of 20/sec, and each velocity command is sent only once. In this UDP-based approach, where packets can get lost or discarded on reception due to corruption, an error correction and order detection scheme is employed, to avoid having to send each command packet multiple times for redundancy.

Each velocity command packet essentially contains an NMEA-0183-compliant[1] sentence as follows:

```
$PVEL,<time_count>,<x-axis value>,<z-axis value>*<checksum>
```

The <time_count> field is an auto-incrementing unsigned integer, used as an index, to allow the platform to determine if a packet is received in-order (hence, valid) or out-of-order (in which case it should be discarded).

The two float velocity values, each in [0,1], are transmitted in their string representation, using 5 decimal digits, which is sufficient accuracy for our needs. The checksum is calculated per the NMEA-0183 specification, and is two characters (hex representation of a byte). Thus, each UDP velocity command packet is 26-36 bytes long (depends on the size of the <time_count> field), and has a 28 bytes header, for a total max size of 64 bytes. At 20 such commands per second, the required bandwidth would be of the order of ~10 Kbps. As in the case of the platform's Advertisement module, this does not include link layer overhead, but again the amount of data rate required does not pose a problem when using a multi-Mbps link like 802.11.

§4. The Display Modules

Each of the display modules is a GUI application that implements specific functionality. The current implementation represents updated code, conforming to the architecture of Figure 5.3: The Control Station Architecture. it has superceded the previous GUI code, which combined all display functions into a single code module, making its mainte-

1 https://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp. See also the (more informative) wikipedia entry at https://en.wikipedia.org/wiki/NMEA_0183.

nance and further development rather arduous.

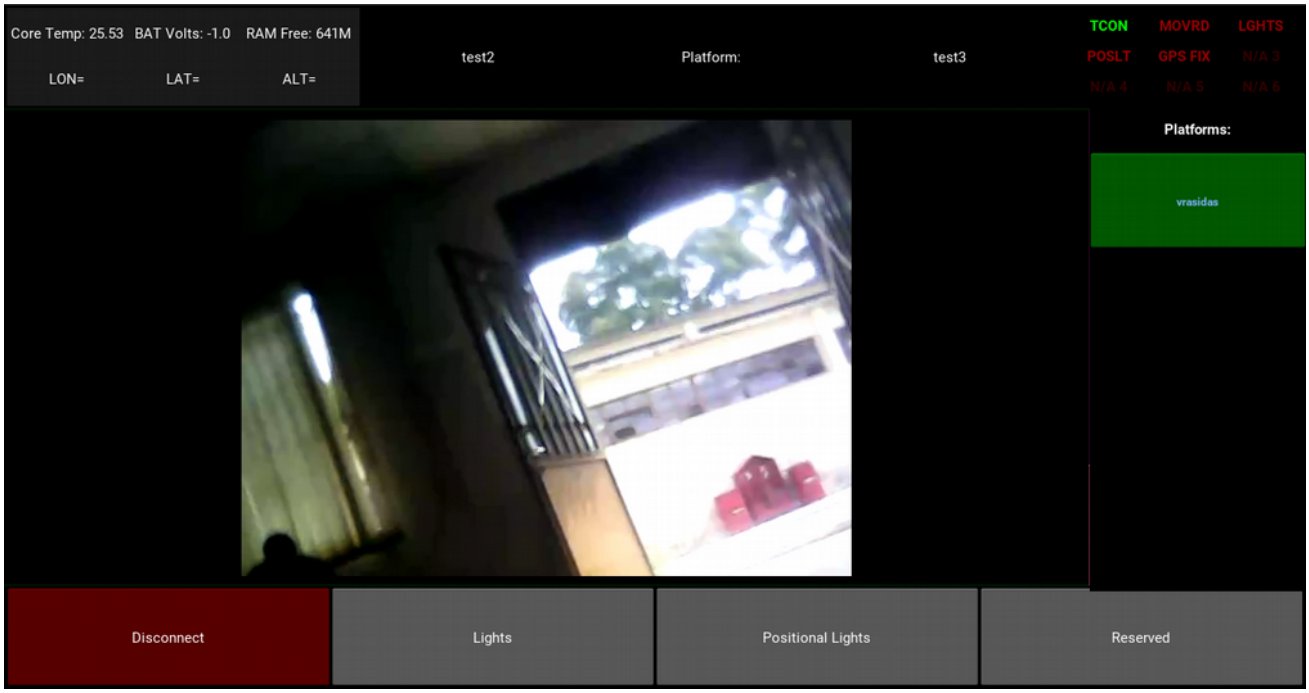

Figure 11.3: The Control Station's Display Modules Layout

The Control Station currently uses kivy as its GUI framework. Kivy is an open-source Python library for native user interface development, that runs, among others, even on Android. The framework takes advantage of GPU acceleration where available, and is maintained to be up to date – the latest version as of this writing is 1.10.0, released concurrently with this work (May 2017). Kivy uses a script-based notation to define the layour of GUI components, then applies pure Python goodiness to manage the component interactions. It is well documented, and [49] has all the details and tutorials one needs.

Using Kivy, the design team has defined three GUI modules, as shown on Figure 5.3: The Control Station Architecture. Each module is also a ROS node.

The screen layout of the Display modules is shown on Figure 11.3: The Control Station's Display Modules Layout.

The top area is the Status Display module, which is basically nothing more than a passive panel filled with indicators and value displays. This panel does not respond to any user input, and only updates the values and indicators it displays when a connected platform transmits its telemetry.

The middle area is the Stream Display module. When the Control Station connects

to a platform, this window displays the feed from the platform's NavCam, and also the list of all platforms found online, with each platform shown as a clickable button – clicking the button connects to that platform.

On the Figure, the Control Station is connected to the network, so it has found the robot, and it is also connected to it, with the NavCam stream being displayed.

The bottom area is the Control Interface module. Currently it contains very few control buttons (only "Disconnect" and two toggles for the platform's lights). As the robot's functionality becomes consolidated, this panel is bound to have additional control options integrated. It is assumed a touch display will be used, eliminating the need for a mouse, and relying on the keyboard and XBox controller for all control options.

4.1. The Stream Display Module

The Stream Display Module displays the NavCam feed from a connected "Brasidas". Internally it uses Gstreamer to receive the transmitted stream, decode it, and render it on the window. Again, a custom Python class is used here, called PiGstVideoInXID (a concrete descendant of the PiGstPLBase class). The implementation takes advantage of the ability of Gstreamer to render the video into a custom window. The `xvimagesink` element accepts a parameter (aptly) named 'xid', which is the XWindow ID of the target window. As can be expected, this method only works on the X window manager. By retrieving the X-ID of the target window, and setting it as the value of `xvimagesink`'s 'xid' parameter during pipeline initialization, the Gstreamer pipeline renders the resulting video directly onto that window.

```
From suboprocess import check_output
.
.
.
# get the xid of the video feed window
#get the xwininfo output
xwininfo_output = check_output(['xwininfo', '-name', 'VC2S Video Feed Panel'])
xid_startidx = xwininfo_output.index('Window id:') + len('Window id:')
xid_endidx = xwininfo_output.index('\"VC2S Video Feed Panel\"')
self.win_xid = int(xwininfo_output[xid_startidx:xid_endidx].strip(),16)
```

Listing 5: Retrieving a Window's XID on Linux

Listing 5: Retrieving a Window's XID on Linux displays the Python code snippet from the Control Station code that retrieves the XID of the Stream Display window. The code takes the output from the `xwininfo` system application and parses it to extract the XID, which is displayed as a hexadecimal value (hence the '16' parameter to the `int` casting of the extracted string at the last line).

The only drawback to this is that Gstreamer cannot be limited to a specific part of the window; it renders over the entire window and any graphical controls it may contain. Thus, a window receiving a Gstreamer video cannot be used for anything more. Of course, if a video is not rendered, the window can function normally.

This drawback can be mitigated somewhat by processing (resizing/scaling/etc) the video in the pipeline, using additional Gstreamer elements such as `videoscale` and `videobox`. [50] has a lot of similar examples. It uses the now obsolete Gstreamer 0.10, however the examples can be adapted to Gstreamer 1.0 with minimal work.

The Stream Display module is actually two separate windows; one that receives the list of discovered platforms, and presents it as a set of clickable buttons, one for each platform (to facilitate choice of platform for connection), and the main window that displays the streamed video. The stream is of course displayed only when the Control Station is connected to a platform; at other times, this window is blank.

In Phase II, the platform selection screen could display the pre-loaded or current area map, then draw the discovered platforms as pins or icons over each corresponding platform's location, updated in real time as platform advertisements are received. The icons can be clickable to allow selection – and perhaps right-clickable to present a context menu with appropriate options. The exact UI layout is still not finalized, as it also needs to be made flexible enough to accommodate possible different options of the various possible payloads.

4.2. The Status Display Module

The Status Display Module is used only to display information and indicators regarding the status of several plarform options, such as whether the lights are on on off,

the charge state of the battery, etc. It serves as the primary visual feedback source for the operator regarding the platform's status. Otherwise, this module does not accept any user input. It is the equivalent of the Annunciator Panel (Master Caution) found on aircraft cockpits, albeit much simpler and less cluttered with readings (for the time being, at least).

Depending on the notifications received, the module's panel can flash warning messages to draw the operator's attention when a critical situation occurs, such a state of low battery charge. The use of audio warnings is also being considered if their use proves justified.

This module also displays all received telemetry values, such as the System Core's temperature (measured via the onboard IMU's temperature sensor), or the amount of available RAM.

4.3. The Control Interface Module

This module provides a series of GUI controls beyond the teleoperation-specific physical controls of the teleop console. The 'Disconnect' button is found here, among others, enabling disconnection from a platform. Other button controls currently include the light switches (both main and positional), and a few placeholder buttons, whose use is yet to be determined. Much like the Status Display module, this module is the result of code reorganization and is not yet complete. The design team is considering the option of transferring to this module some amount of functionality that were implemented as physical controls on the (old) Arduino-based teleop console.

CHAPTER 12 .
Testing and Verification

Testing was subsumed into the development process since the early stages of the project. Every subsystem and code module was tested individually for errors and performance. However, testing the prototype as a whole proved a bit more difficult; multiple hardware and software components had to be completed and integrated before the entire prototype could be put to the test.

§1. Network Tests

Given that the network infrastructure is taken ready with no modifications or additions, little dedicated testing was implemented. The network performance was tested as part of various general system tests, where the platform was teleoperated in increasingly larger distances from the Control Station, until the connection was lost.

As has been mentioned earlier, tests on software versions using `motion` for video streaming, recorded network utilization in excess of 5 Mbps, while Gstreamer-based versions ran at 1.6-2 Mbps. These were taken off the OpenWRT router's bandwidth monitoring page, and are tranfer rates, i.e. including the Layer-3 and higher protocol overheads, but not the physical and data link framing overheads. The fact that 802.11 has a rather high framing overhead is an open topic of discussion among the design team members, but replacing that with a wireless implementation of more efficient protocols is not being considered at the moment.

All such tests took place in the ARTC. The place is filled with buildings, vehicles (metal obstacles), and multiple WiFi access points (interference) operating right outside the area. During those tests, the network permitted effective teleoperation (using `motion`) to a maximum recorded distance a little over 250 m, with a large building interposed between the platform and the Control Station. The network utilization was also monitored closely to determine the amount of bandwidth that the platform-Control Station link requires. This experimentally recorded range limit demonstrates why `motion` is inadequate,

as with a lower bandwidth usage the robot would most likely have ventured a bit further. However, no test has yet taken place using the newer, Gstreamer-based implementation. It is expected that the effective teleop range will be significantly longer, though.

A second, limited test was performed as part of a presentation held at the Military Academy of Athens. The Control Station and Access Point were positioned inside the Academy's main amphitheater, and the platform was teleoperated to a course leading it outside the amphitheater and the building. The software during this test did use Gstreamer, but the test was terminated while the robot was approximately 50 m from the Access Point (straight-line distance), and until then the connection was active and the video feed and teleoperation were smooth. No attempt was made to move the platform further away to determine the range limit under such conditions. However, attenuation was quite severe, as there were several attendees (and thus active smartphones) in the intermediate area, not to mention that the signal had to pass through a concrete wall.

§2. Platform Software Tests

Most of the platform software tests were run on the RPi. A few simple pieces of code were tested first on a development laptop, then deployed to the RPi, but the majority of the code was debugged and tested on the platform itself. Especially those modules that facilitated hardware access could be tested on no other machine.

Apart from correctness of code, the tests sought to measure primarily the amount of RAM and CPU utilized by each module, as well as whether each started and shut down appropriately and without exceptions or possible memory leaks.

These tests showed that the current implementation runs with no problems on the RPi 3B. Memory suffices with no swapping involved. During video streaming no CPU core went above 40% utilization for an appreciable amount of time. By comparison, tests run on the older, `motion`-based versions, recorded CPU utilizations of the order of 40-60% on two cores, a result of `motion`'s software MJPEG encoding. RAM utilization was also greater in the `motion`-based versions, reaching a peak of almost 550 MB of RAM,

versus 350 or so MB in the newer version.

The most significant difference was observed in overall video quality. Despite the lower bandwidth, the H.264-encoded video has better quality than `motion`'s MJPEG, since to reduce bandwidth the design team had set the MJPEG quality to a rather low setting. The hardware H.264 encoder available on the Videocore-IV GPU has a default encoding setting of 'H.264 high-profile'.

§3. Control Station Software Tests

As currently the Control Station software runs on the laptop on which it is also developed, only a few dedicated test runs were made; every functionality not requiring platform connectivity could be tested at any time – and indeed was – by simply running the code.

The visualization modules, and in particular the GUI component layout, were tested separately. A mock Kivy module was written, which rendered the windows and controls according to how it was intended each time, but the controls had no user responsiveness or functionality. This way, the Kivy script additions and modifications could be tested for correctness without mixing them with already existing, complex code. It made error tracing far easier. Whenever the layouting code and script tested through the mock module was deemed satisfactory, it was copied over and integrated with the rest of the Control Station code base.

The focus of testing for the Control Station code was on responsiveness. Performance was not considered until recently, since it was assumed that a desktop-based system would be used for the Control Station. After the decision to use a single-board computer in that subsystem as well, future tests have been scheduled to test the Control Station software's performance as well, since the GUI puts a heavy strain on embedded system resources, especially RAM – and RAM on the Pi is limited.

These tests will heavily influence the further Control Station development. If the tests show that, with the current single-board computer capabilities, this solution is not viable, this approach will be re-evaluated, and other solutions will be examined. There al-

ready exists a proposal to use a mini-itx-based computer board, such as those typically used in hi-end car PCs. Such a system could easily have a mobile, lower-power version of a Core-i5 and several GB of RAM, allowing extended capabilities for the Control Station.

A final aspect of the Control Station that needs to be tested is operability. As thus far the controls have been operated by the members of the design team, who have intimate knowledge of the system, no feedback exists on what sort of window layout and physical controls is easier and more intuitive to operate, as far as a person who has not participated in the project development is concerned, and who most likely will not have an engineering background (i.e. any one of the expected future operators, but not a designer).

At this stage in development, the most prominent control layout concept involves a set of physical and software controls akin to the interface of a first-person shooter game (at least for control a single platform), where the main view comes from the NavCam, and an XBox-compatible game controller is used to steer the vehicle and handle control of the payload (turret azimuth/elevation, fire button, etc.). An additional custom keyboard panel will most likely be assembled, to permit extra function-specific buttons, instead of mapping all functions to controller keypresses.

CHAPTER 13 .
Interlude II

Programming a robot is no easy task, even a simple teleoperated one like the "Brasidas" Mk-I. Again, the reader is reminded that the streamlined form of process appearing in the chapters of Part II is convenient, but not representing reality. In truth, things did no go so easy and the path was never very clear.

Of course, it must be emphasized that the design team had next to zero experience in the matter. This whole process has been an entire 'learn through trial-and-error' travel down failure road. In fact, it was mostly 'learn through error'.

Just sorting out what OS distribution and version to use took over three months. Initially, the design team settled on ROS Indigo, which was available on Ubuntu ARM, but not on Raspbian Wheezy. An attempt to compile it took a few days of effort and resulted in a not-so-fully-functional ROS install. So for a while, Ubuntu 14.40 (armhf) was used on the Raspberry Pi. When Gstreamer was found to be applicable, things went south again. Now Gstreamer may be available on both Ubuntu and Raspbian, but – most importantly – the OpenMAX-based pipeline elements were available only for Raspbian Jessie, and not Ubuntu ARM. Thus, after a lot of deliberation, the OS went back to being Raspbian, this time the more recent Jessie distribution. Thankfully, in the second iteration the desing team were wiser, and attempted to compile a leaner ROS setup, which succeded – the release was even upgraded to use ROS Kinetic!

The previous software versions, which relied too heavily on Pyro4 and `motion`, are also good examples of bad development; the latency of `motion`'s feed and the jerking and rubber-banding of teleop control through Pyro4 should have resulted in the immediate drop of that implementation. However, due to time constraints and focus on attempting to close Phase I and move into Phases II and III, in trying to implement autonomous navigation, caused the problem to be sidelined and tolerated, until it became apparent after some tests that that implementation was going to no serious town. It was a good waste of time and a good lesson learned.

Even Kivy, the GUI framework, was not the project's first choice. PyQT was initially used, but proved cumbersome and resource-demanding. A different framework, VisPy[1], was then adopted. This framework was still under development, and suffered from a multitude of bugs. Eventually, it was abandoned in favor of Kivy.

Another issue was that of the Control Station control layout. Initially, a custom controller console was constructed, using an Arduino board and compatible joystick, which was programmed accordingly and used alongside a standard keyboard. The console sent sentences formatted following the NMEA-0183 HS (High-Speed) protocol, except the sentence headers used custom symbols, and not two-letter vendor or sentence codes. A sample sentence is shown below (the '0' and '1' entries are button states, the other four numbers are axis positions, the last value is the sentence checksum):

```
$TCSTA,1,0,1,1,412,887,515,512*57
```

Although this approach was very flexible and extensible (we could add as many buttons, extra joysticks, and additional controls are we liked), it used a custom board instead of a COTS solution, and consumed a non-trivial amount of development time. Most of it was spent in designing the Arduino communications protocol. The teleop console was designed to initially accommodate an old analog 4-axis, 4-button joystick (the author had one lying around from his early Wing Commander 2 and 3 days, and contributed it to the project), although eventually a different analog joystick, a 2-axis Arduino-compatible one, was used.

Another idea considered the use of a gamepad (XBox-compatible game controller). This can in theory aggregate more controls in a smaller package, and make the Control Station more portable (perhaps even man-portable), but no such controller was available initially, and there were concerns that it may be less convenient than the keyboard+joystick approach.

Eventually, both options were tested. Our concerns about the XBox controller proved to be inaccurate, and thus the XBox option was adopted. It is by far the most convenient option, as it allows us to use a COTS component in place of a custom-made

1 http://vispy.org/

one. Plus, the keyboard still remains in use, and is a viable option for providing additional controls.

The combination of multiple architecture revisions and lack of knowledge on the design team's part have resulted in slow development progress, and a Phase I that has taken more than it should, and thus far delivered less than expected. Finally however, things are beginning to coalesce into something that will resemble a robot… eventually…

EPILOGUE

There is still much work to be done. Phase I may be practically complete, but "Brasidas" is not – plus, you know what they say about "almost": it only works when throwing horseshoes and hand grenades.

The software has been designed to be modular and plarform-agnostic, opening up the possibility of using it on other ground vehicles. Thus, it can form the core software of a family of vehicles that may perform totally different functions, but be able to interconnect and communicate, all the better to execute their assigned mission.

The development speed is likely to pick up from now on, as a valuable knowledge base of know-how and practical skills has been developed.

While the intent is to eventually have "Brasidas" operate autonomously, a robust R.O.V. implementation is required first. A R.O.V. includes all low-level functionality and basic concepts that will be required by the higher-level protocols employed in autonomous operation. In addition, many aspects of autonomous operation are at the moment still being researched actively, and lack a robust implementation. Not many aspects of such implementations are standardized, which means any autonomous operation architecture we might implement is likely to run into standardization issues (non-conformance to standards) in the future. Thus, it is advantageous to the project's development to first implement a R.O.V. version that is pretty standardized, then build upon this and relevant standards when developing autonomous operation elements. This approach will reduce development and cost risks substantially.

§1. Future Work

"Brasidas" is a project as simple as a remotely operated vehicle. It can also be as complex as a fully operational military-grade autonomoung ground robot. The work outlined in this dissertation has layed the foundation for turning this project into serious business. As Phase 1 is all but completed, and Phase 2 begins, a multitude of tasks be-

come available.

A robust SLAM implementation for day/night outdoors environment is required, which can be a research project on its own.Visual SLAM seems promising, but it requires night vision cameras at the least, to satisfy the nocturnal operation requirement. Other spatial sensor configurations, like IR/ultrasonic arrays and 3D laser scanners might be able to fill the void if a night vision camera is not available, but these are active sensors that can potentially give out the robot's position. A fair amount of research and experimentation is required on the SLAM matter, and then the specific requirements for this component must be drawn up so design and analysis of the component's implementation can finally begin.

Autonomous operation will require a specification for the mission logic and task execution algorithms, as well as autonomous navigation. A robust implementation for each of these is also needed. There are several tried and proven solutions for autonav, but autonomous operation is a field of robotics and AI where few mature choices exist at the moment (May 2017), so an effort must be made to test and evaluate the possible options, and perhaps combine and improve them until the result is satisfactory. Once a suitable solution is identified, system requirements for the autonomous operation and navigation must then be specified before design and implementation of this component must begin.

All the above work has been outlined roughly when the project workload was divided into Phases, but this is not just a problem of how and when to implement a solution, but ***what*** solution to implement, making the matter more a subject of scientific research than evaluation of available options.

As if all the above are not enough, "Brasidas" also needs to produce alternate payload packages, to fulfil the operational needs of potential users. Each payload requires investigation and study into potentially different fields, and each can be a separate project on its own.

All these of course assume that at some point, "Brasidas" becomes a 'proper' research project, backed by access to required restricted equipment and facilities, and basic

funding. But that is another story...

§2. Survivor's Guide to Robot Software Design

This short text presents the design team's lessons learned thus far. It is a compilation of 'words of wisdom'. Though it is not yet complete, like the project itself, it is hoped that it will help others not make the same mistakes. The guidelines apply to any well-written software, so it is likely that they will be encountered elsewhere as well.

This part only contains software-related guidelines. Cpt (AA) Katselis' thesis presents the part containing the hardware-related guidelines.

1.0. Get a Rounded-Out Design Team

You need a software expert (programmer), an electronics expert (electrical engineer), and a mechanic (mechanical engineer) **at the very least**, before embarking on a quest to design and build a robot. The more relevant an experience they have, the better. Do not even start until you have those.

Ideally, your robotics "adventuring party" should also include an end-user liaison (to provide problem-domain feedback from the end user's perspective), a physicist (good all-round helper boy, ocassionally pops a weird but useful theory or two, do not assign him very specialized tasks), and a marketing expert (to monitor expenses and funding usage, and also to organize the product presentations and marketing campaign, and perhaps conduct market research on potential business opportunities).

1.1. Know What You Are Making

Figure out your vision as early as possible, before even determining whether it is viable technologically or not. A clear objective allows you to easily determine what needs and need not be done, to evaluate requirements and proposed solutions with greater certainty, and to more easily avoid being distracted down a path that will not deliver. Write a scenario, create a CGI video, draw a cartoon, or anything that will allow you to form a clear picture of what you want to make.

Sun Tzu's advice applies here in its entirety: "If you know yourself and your enemy, you need not fear the result of a hundred battles."

1.2. Before You Do It, Find Out If Someone Else Has Already Done It

This point applies generally to all kinds of software projects. Robotics in particular is a hot subject, and many research teams are working on solving the problems in the field. Much code is produced and disseminated as part of that research. It may not be industrial-quality code, but is is ready code. Before attempting to re-invent a wheel, spend some time searching if someone else has already invented the wheel you are after. If they are, and you can use their code, do so.

Even if you find ready code, ALWAYS make sure it fulfils the requirement YOU were trying to satisfy **precisely**. Also, make sure it is **optimized**, and not low-grade research code. This cannot be emphasized enough. Robotics covers a vast spectrum of applications, and each one imposes different requirements. It may look similar, but make sure it is similar enough, and will not cause you headaches in the long run.

Robotic applications are often demanding in terms of RAM and CPU. You don't want to adopt code into your project that wastes these two very valuable computer resources, or that you will have to later optimize yourself. Optimizing code is one of the hardest missions a programmer can take, and optimizing code that is not your own incurs a forbidding time cost.

If there are differences but are small, evaluate whether modifying the foreign code is worth the time, or an in-house implementation is a better option. Also, sometimes you can get away with murder and use bad quality code if it is working – see the next guideline.

1.3. Be Flexible and Keep an Open Mind

Never reject an idea or possible solution until you have evaluated it against your project's requirements, regardless of whether it seems fitting in concept or not. In any bleeding-edge research field like robotics, there are no "correct" answers; each re-

searcher's perspective is unique and can provide useful insight to the problem at hand, assuming of course someone else has not already figured the answer (see the previous paragraph). Abstract the specific problem to generic requirements, and only evaluate everything against the requirements, not the problem itself.

As an example, consider what is the best propulsion solution for a ground vehicle that has to move over thin, soft mud: wheels or tracks? Does it have to do with overall vehicle weight? Speed?

What do you mean it didn't occur to you that in the Everglades they use airboats?

Remember Polyvios Dimitrakopoulos's words in 'The Iron Will': "Man rarely comprehends what he sees; often he sees what he comprehends." Avoid comprehending first and seeing after.

1.4. Use a Results-First Approach to Coding

Adopt a choice of programming language and development process that delivers usable results as soon as possible. Avoid pondering code optimization until after you have a complete program or application component. That way you can reject bad choices early on and with little cost in time and effort.

Unless you have a short deadline to meet (a circumstance you generally should try to avoid getting into) test every bit of code whenever it is complete enough to permit testing. It is easier to spot an error in 100 lines of code, than it is in 1,000. Then, after you write the next 100 lines of code, do test the previous code again, on occasion. You never know.

This is general software engineering advice, but it cannot be emphasized enough. Really.

1.5. Watch Out for the Dreaded Comms

Unless aiming for a totally autonomous or wired-connections system, communications is the real deal-maker or deal-breaker during the design and manufacture of a robotic system. The communications scheme that is to be adopted should satisfy the fol-

lowing obvious (and not-so-obvious) requirements:

– Its absolute minimum required data rate should be estimated as late in the project development as possible. Only employ a comms component when really needed, and then use an option that fully covers the need. By delaying selection and adoption of a comms component, the designer allows as many details and needs to become apparent, as possible.

– It should afford a data rate at least TWICE the estimated one, **including** future expansion plans, to account for the worst case of overhead and link degradation due to range.

– ALWAYS test the candidate comms scheme out to the specified maximum range, in as realistic conditions as possible. If it fails to deliverer as advertised, determine if it's worth considering a revision, and if not, discard it.

– IP-based communications schemes afford the biggest flexibility in system integration, but also have significant data overhead. However, almost all the IP/OSI-7 protocol implementations can be found as industrial-quality, fully-certified, open-source code, making the engineer's job much easier. DO NOT adopt a non-IP-based scheme, unless absolutely necessary.

– Adjust all range and data rate requirements according to the MAXIMUM expected number of concurrent users/connections during an actual field operation of the system. Then, further increase the resulting values by 20% (designer's sanity margin).

– Constantly investigate the latency-vs-network load relationship, and determine whether the communications scheme can fulfil timing requirements along the entire range of its expected data rate utilization.

1.6. Your Electrical Engineer is Your Best Friend

Robotics needs both software and hardware. And the software often runs close to the hardware. As hard as it seems for you to write code that runs on certain hardware, it is equally hard (pun intended) for the electrical engineer to devise hardware that will run

your code.

Make sure you communicate with your team's electrical engineer often, to early on clarify on any issues regarding low-level protocols and capabilities of the hardware used. Top performance requires **both** hardware and software optimizations, and often your (software) requirements will become constraints on the electrical engineer's hardware choices, and vice versa (for instance, if you want H.264 encoding, the electrical engineer must choose a computer board that features hardware H.264 acceleration). Gaming consoles use hardware that is often much worse than the best desktop PC configuration, yet they deliver far greater performance. They achieve this because they optimize at both the hardware and software layers. For example, they avoid if-based error checking at certain points in code when they know that due to the hardware choices that error or disparity will never occur.

1.7. Your Mechanical Engineer is Your Second Best Friend

Your software controls a mechanical vehicle, and the electrical engineer's hardware is also mounted on that vehicle. Coordinate with the team's mechanical engineer on matters of component placement and dimensions. You need to be aware of the propulsion (drive and steering) method chosen, to properly structure your Transmission Interface software. You also need to know where on the platform the various sensors are placed (coordinate transformations).

Software is less tightly coupled to mechanical parts than it is to electronics hardware, but you and the mechanical engineer still need to see eye to eye on project decisions and implementation details.

1.8. Never Believe Other Robots' Specifications

Most designers will publish results that are slightly… exaggerated, either (rarely) to satisfy their ego, or (mostly) to fool competitors into wasting time and effort surpassing a benchmark that is usually unattainable without exorbitant cost. If the specification of a RPi-based robot with a HD webcam says it can do real-time pedestrian recognition and

point cloud-based mapping, grab your popcorn and read the rest of the specs in a kidding mood, because it most likely **is** kidding. Unless you have discovered Alan Turing's reincarnation.

The same guideline should apply to your selection and adoption of third-party software libraries, especially when they are non-commercial and their license states they are provided 'AS IS'. Actually, the development of "Brasidas" thus far has been rife with such examples of gullibility on the design team's behalf.

§3. In Closing

"Brasidas" is an ambitious project, especially given the restrictions regarding funding and access to equipment. It represents but a minuscule part of a vision; in particular the part of the place of robotic software and systems in the ground battlefields of the 21st century.

Many more visions have come before this, and much grander. This is not a better or worse vision, it is just another one, perhaps different, perhaps not. It will not be the vision to rule them all, but it could possibly contribute something to what is to come. Maybe soon some scientist will discover how to create true AI, or something equally ground-breaking. Even if that does not happen in our lifetimes, the proliferation of automated, robotic, and autonomous systems will happen.

Make no mistake, robotic systems are here to fight, and are here to stay. Soldiers and defense designers alike must adapt to this reality, or become extinct – the former literally, the latter professionally.

It sure has been fun, though…

ANNEX A: Installing Gstreamer 1.0 OpenMAX Extensions on Raspberry Pi

All versions of Raspberry Pi feature the VideoCore-IV GPU, which among others includes a set of multimedia codecs in hardware. These modules can be accessed by using the OpenMAX extensions[1]. Gstreamer includes pipeline elements which can take advantage of these extensions, to do hardware-accelerated encoding or decoding of media.

On the Raspberry Pi, these pipeline elements are packaged in gstreamer1.0-omx, but this package is available only for Raspbian Jessie, and the latest Raspbian Wheezy distributions. If you have an older RPi that still runs Wheezy and the distribution is out of date, you need to either upgrade the distribution, or attempt to build gstreamer1.0-omx from source.

Note that this Appendix is only concerned with the new Gstreamer release 1.0. The older 0.10 release that was used for a long time, can support omx as well, but it is no longer supported, so if you are setting up a Gstreamer system now, it is recommended that you use version 1.0. Before installing anything, always do a

```
pi@raspberrypi ~ $ sudo apt-get update
pi@raspberrypi ~ $ sudo apt-get upgrade
```

to make sure all repositories and dependencies are up to date.

If you run Wheezy, also upgrade the distribution to the latest version

```
pi@raspberrypi ~ $ sudo apt-get dist-upgrade
```

Then, all you have to simply do to get gstreamer installed along with the OpenMAX extensions, is to issue

```
pi@raspberrypi ~ $ sudo apt-get install libgstreamer1.0-0 \
        gstreamer1.0-{omx,alsa} gstreamer1.0-tools \
        gstreamer1.0-plugins-{bad,base,good,ugly}
```

This should also automatically pull in all additional package dependencies, and all should be installed after a while. To verify that the omx pipeline elements are indeed in-

1 Official site: https://www.khronos.org/openmax/

stalled, do a

```
pi@raspberrypi ~ $ gst-inspect-1.0 | grep omx
```

and you should get a result that includes all or part of the following lines:

```
omx:  omxhdmiaudiosink: OpenMAX HDMI Audio Sink
omx:  omxanalogaudiosink: OpenMAX Analog Audio Sink
omx:  omxh264enc: OpenMAX H.264 Video Encoder
omx:  omxvc1dec: OpenMAX WMV Video Decoder
omx:  omxmjpegdec: OpenMAX MJPEG Video Decoder
omx:  omxvp8dec: OpenMAX VP8 Video Decoder
omx:  omxtheoradec: OpenMAX Theora Video Decoder
omx:  omxh264dec: OpenMAX H.264 Video Decoder
omx:  omxh263dec: OpenMAX H.263 Video Decoder
omx:  omxmpeg4videodec: OpenMAX MPEG4 Video Decoder
omx:  omxmpeg2videodec: OpenMAX MPEG2 Video Decoder
```

You might get less packages than the above listing; this is usually because in latest releases some (notably the audio sinks) are disabled, since they are available directly at the kernel level via ALSA (so you can use them just as well by using the Gstreamer alsa-related plugins).

All the extensions but one are decoders, but there is an H.264 encoder, and that is what most folks are after. Note that despite H.264 requiring a licence[1], the purchase of a RPi includes this licence, so anyone purchasing a Pi can use the H.264 encoder with no legal worries.

You can test a successful installation by connecting a USB webcam on the Pi (will be assigned node `/dev/video0`) and running in bash the following pipeline (change the 'width', 'height', and 'framerate' parameters to a resolution supported by your camera):

```
pi@raspberrypi ~ $ gst-launch-1.0 -ve v4l2src device=/dev/video0 \
    ! video/x-raw,width=320,height=232, framerate=15/1 \
    ! omxh264enc ! rtph264pay config-interval=3 \
    ! udpsink host=<IP_of_target_PC> port=15575
```

Then, on the computer having the IP specified in the 'host' parameter above, run the following pipeline to receive the stream:

```
pi@raspberrypi ~ $ gst-launch-1.0 -ve udpsrc port=15575 \
   caps="application/x-rtp" ! rtph264depay ! avdec_h264 \
    ! xvimagesink sync=false
```

You can of course use any port you wish. If all works as intended, a window should

1 Also check Cisco's OpenH264 (http://www.openh264.org/) for an open-source version of H.264.

pop up, showing the camera video, and with minimal latency. Beyond this, all pipeline elements have additional parameters that you can tweak to tune the streaming to your liking.

As a side note, the author has successfully installed and used these elements to stream H.264-hardware-encoded video, at 640×480@30 fps from an old RPi Model A+ (only used ~60% of the single-core CPU). This was possible because of a third-party repository that offered precompiled gstreamer1.0-omx binaries for the older Wheezy releases. However, with the inclusion of gstreamer1.0-omx into the latest Wheezy and Jessie releases, this repository has not been updated since late 2014, and its key has expired. This process is detailed in this (outdated) RPi forum topic:

https://www.raspberrypi.org/forums/viewtopic.php?p=293634

Finally, if all else fails, you can attempt to build from source as a last resort. The following post in a Stack Exchange topic explains the process:

https://raspberrypi.stackexchange.com/a/4628

Note: The author assumes NO responsibility for bad source building attempts. Be warned that attempting to build Linux packages from source can result, among others, in a deep knowledge of C++ and linux kernel programming, fascination with compilers, obsession with instruction optimizations, lack of social contact, decrease in vocabulary, possibly slight overall increase in intellect, a fondness for pizza, burgers, and all kinds of junk food, and a massive collection of single-board computers and electronic boards of all kinds that takes up space and serves no purpose.

ANNEX B: Forwarding Broadcast Packets through an OpenWRT Router with `socat`

The platform discovery functionality required that platform broadcast a basic set of parameters, that enable Control Stations to connect to them. The information also includes a basic status report on the whereabouts and condition of the platform. This enables a quick overview of the platforms' state without having to connect to every one of them.

This information is carried in broadcast UDP packets transported over port 21000, but you can use any port you prefer. Simply change the '21000' below to whichever port number you select.

Since both the platform and Control Station computer actuals sit behind their respective network routers, broadcast packets are normally prohibited both from entering the "Brasidas" network out through the platform router, and entering the Control Station LAN in through the Control Station router.

The routers used when this work was prepared (May 2017) run the OpenWRT firmware. After several failed efforts to configure the router firewalls into forwarding broadcast packets, a workaround was implemented using `socat`. `socat` is by default not a part of a pre-compiled OpenWRT image, so it needs to be downloaded first. This can be accomplished by first logging in to the OpenWRT router via `ssh`, and issuing the following commands:

```
opkg update
opkg install socat
```

The AREDN™ mesh OpenWRT distribution does already include `socat`, so there's no need to install it if you use that distribution.

Then login to the router, either via the graphical interface (LuCI), or via ssh. If you use the graphical approach, go to System→Startup. There, scroll down to the end of the

page, where the contents of the file `/etc/rc.local` are displayed. This file contains shell commands that are executed at the end of the boot process. If you login via ssh, you need to edit the file manually, using `vi` (which is the only editor included in an OpenWRT distribution). Before the line '`exit 0`', enter **one** of the following two lines, depending on which router you are configuring. Note that although the solutions are given on two lines here (they can't fit on the page otherwise), you should type each in a single line, without the '\' symbol (which stands for 'line continuation').

For the router installed on the platform, which needs to forward broadcast packets from the internal LAN to the mesh side, use the line:

```
socat UDP-RECVFROM:21000,broadcast,bind=<LAN broadcast>,fork \
      UDP-SENDTO:<mesh broadcast>:21000,broadcast
```

where `<LAN broadcast>` and `<mesh broadcast>` should be replaced with the LAN-side broadcast IP address and mesh broadcast IP address respectively.

For the router installed on the Control Station, which needs to forward broadcast packets from the mesh to the internal (Control Station) LAN, use this line:

```
socat UDP-RECVFROM:21000,broadcast,bind=<mesh broadcast>,fork \
      UDP-SENDTO:<LAN broadcast>:21000,broadcast
```

In order for the `socat` workaround to work, the Control Station router's firewall should be configured to allow port 21000 connections to the router (i.e. as INPUT, not FORWARD).

REFERENCES

1: *The Inside Story of the SWORDS Armed Robot "Pullout" in Iraq*, http://www.popularmechanics.com/technology/gadgets/a2804/4258963/ (accessed 24 Apr 2017)
2: Grady JO, *System Requirements Analysis, 2nd Edition*, Elsevier Inc., 2014
3: National Research Council, *Technology Development for Army Unmanned Ground Vehicles*, The National Academies Press, 2002
4: *Robot patrol: Israeli Army to deploy autonomous vehicles on Gaza border*, http://www.foxnews.com/tech/2016/09/01/robot-patrol-israeli-army-to-deploy-autonomous-vehicles-on-gaza-border.html (accessed 25 Apr 2017)
5: *An Israeli hybrid robot hauls 1.2 tons on high risk missions*, http://defense-update.com/20140611_an-israeli-robot-hauls-1-2-tons-payload-on-high-risk-missions.html (accessed 30 Apr 2017)
6: Shelly G, Rozenblatt H, *Systems Analysis and Design, 9th Edition*, Course Technology, 2012
7: Dennis A, Wixom BH, Roth RM, *System Analysis and Design*, Wiley, 2012
8: Crinnion J, *Evolutionary Systems Development: A Practical Guide to the Use of Prototyping Within a Structured Systems Methodology*, Plenum Press, 1991
9: U.S. DoD, *Military Standard 881C: Work Breakdown Structures for Defense Materiel Items*, U.S. DoD, 2011
10: Carroll, John M., *Making Use: scenario-based design of human-computer interactions*, MIT Press, 2000
11: U.S. DoD, *Systems Engineering Fundamentals*, Defense Acquisition University Press, 2001
12: Boehm B, *A Spiral Model of Software Development and Enhancement*, 1988
13: Boehm, B, *Spiral Development: Experience, Principles,and Refinements*, , 2000
14: *Unattended Ground Sensor*, https://en.wikipedia.org/wiki/Unattended_ground_sensor (accessed 24 Apr 2017)
15: Feickert A, *The Army's Future Combat System (FCS): Background and Issues for Congress*, Congressional Research Service, 2009
16: Bruegge B, Dutoit A, *Object-Oriented Software Engineering: Using UML, Patterns, and Java, 3rd Edition*, Prentice Hall, 2010
17: Gang of Four, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
18: Steinberg AN, Bowman CL, *Rethinking the JDL Data Fusion Levels*,
19: Holbrook H, Cain B, Haberman B, *Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast*: , https://tools.ietf.org/html/rfc4604 (accessed 27 Apr 2017)
20: ISO/IEC 7498-1, *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*, , 1994

21: NSA, *Commercial Solutions for Classified Program*: *Capability Packages*, https://www.nsa.gov/resources/everyone/csfc/capability-packages/ (accessed 11 Apr 2017)
22: Baird, Leemon C.; Bahn, William L.; Collins, Michael D., *Jam-Resistant Communication Without Shared Secrets through the Use of Concurrent Codes*, U.S. Air Force Academy, 2007
23: Bahn, William L., *Concurrent Code Spread Spectrum: Theory and Performance Analysis of Jam Resistant Communication Without Shared Secrets*, University of Colorado, 2012
24: Nielsen J, *Usability Engineering*, Morgan Kaufmann, 1993
25: Lentin J, *Learning Robotics Using Python*, Packt Publishing, 2015
26: Donat W, *Make: A Raspberry Pi-Controlled Robot*, MakerMedia, 2015
27: *Robot Operating System*, www.ros.org (accessed 14 Apr 2017)
28: Martinez A, Fernandez E, *Learning ROS for Robotics Programming*, Packt Publishing, 2013
29: Goebel PR, *ROS By Example, Volume 1 (version 1.1.0 for ROS Indigo)*, published by the author, 2012
30: Goebel PR, *ROS By Example, Volume 2 (version 1.0 for ROS Indigo)*, published by the author, 2014
31: linux-kernel@vger.kernel.org mailing list archive, *Unix sockets via TCP on localhost: is TCP slower?*: , http://linux-kernel.2935.n7.nabble.com/Unix-sockets-via-TCP-on-localhost-is-TCP-slower-td376893.html (accessed 13 Apr 2017)
32: *AREDN Mesh-enabled OpenWRT distribution*, www.aredn.org (accessed 16 Apr 2017)
33: *IEEE 802.22*, http://www.ieee802.org/22/ (accessed 29 Apr 2017)
34: Flores AB, Guerra RE, Knightly EW, Ecclesine P, Pandey S, *IEEE 802.11af: A Standard for TV White Space Spectrum Sharing*, IEEE, 2013
35: *OpenWRT Embedded Linux-based Router Firmware*, https://openwrt.org (accessed 29 Apr 2017)
36: *HSMM-MESH mesh-enabled OpenWRT distribution*, http://www.hsmm-mesh.org/ (accessed 16 Apr 2017)
37: *DD-WRT Embedded Linux-based Router Firmware*, http://www.dd-wrt.com (accessed 24 Apr 2017)
38: *PYRO (PYthon Remote Objects)*, http://pythonhosted.org/Pyro4/ (accessed 15 Apr 2017)
39: Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, High Performance Computing Center Stuttgart (HLRS), 2015
40: *Motion - a Software Motion Detector*, https://motion-project.github.io/ (accessed 22 Apr 2017)
41: *Requests: HTTP for Humans*, http://docs.python-requests.org/en/master/ (accessed 22 Apr 2017)
42: *Gstreamer: An Open-Source Multimedia Framework*, https://gstreamer.freedesktop.org/ (accessed 21 Apr 2017)
43: NETGEAR Support, *Link Rate and Transfer Speed*: , https://kb.netgear.com/19668/Link-Rate-and-Transfer-Speed (accessed 21 Apr 2017)

44: *OpenSLAM.org: Platform for SLAM Algorithms*, https://www.openslam.org/ (accessed 22 Apr 2017)
45: Martijn Tijndagamer, *A Python module for accessing the MPU-6050 digital accelerometer and gyroscope on a Raspberry Pi*: , https://github.com/Tijndagamer/mpu6050/blob/master/mpu6050/mpu6050.py (accessed 22 Apr 2017)
46: Madgwick SOH, Harrison AJL, Vaidyanathan R, *Estimation of IMU and MARG orientation using a gradient descent algorithm*, Proceedings of the 2011 IEEE International Conference on Rehabilitation Robotics (ICORR) 2011; 1 (152-161)
47: Thrun S, Montemerlo M, Dahlkamp H, Stavens D, et al., *Stanley: The Robot That Won the DARPA Grand Challenge*, 2007
48: Albus et al., *4D/RCS Version 2.0: A Reference Model Architecture for Unmanned Vehicle Systems*, National Institute of Standards and Technology, 2002
49: *KIVY - Open source Python library for rapid development of applications*, https://kivy.org/ (accessed 20 Apr 2017)
50: *Gstreamer cheat sheet*, http://wiki.oz9aec.net/index.php/Gstreamer_Cheat_Sheet (accessed 13 Apr 2017)