View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

# Towards an Open, Distributed Software Architecture for UxS Operations

**"It's difficult to work in groups when you're omnipotent," – Q**
**Star Trek: The Next Generation:: Deja Q (1990)**

Charles D. Cross*, Mark A. Motter†, James H. Neilan‡, Garry D. Qualls§, Paul M. Rothhaar¶, Loc Tran#, Anna C. Trujillo††, and B. Danette Allen‡‡

*NASA Langley Research Center, Hampton, VA, 23681*

**To address the growing need to evaluate, test, and certify an ever expanding ecosystem of UxS platforms in preparation of cultural integration, NASA Langley Research Center's Autonomy Incubator (AI) has taken on the challenge of developing a software framework in which UxS platforms developed by third parties can be integrated into a single system which provides evaluation and testing, mission planning and operation, and out-of-the-box autonomy and data fusion capabilities. This software framework, named AEON (Autonomous Entity Operations Network), has two main goals. The first goal is the development of a cross-platform, extensible, onboard software system that provides autonomy at the mission execution and course-planning level, a highly configurable data fusion framework sensitive to the platform's available sensor hardware, and plug-and-play compatibility with a wide array of computer systems, sensors, software, and controls hardware. The second goal is the development of a ground control system that acts as a test-bed for integration of the proposed heterogeneous fleet, and allows for complex mission planning, tracking, and debugging capabilities. The ground control system should also be highly extensible and allow plug-and-play interoperability with third party software systems. In order to achieve these goals, this paper proposes an open, distributed software architecture which utilizes at its core the Data Distribution Service (DDS) standards, established by the Object Management Group (OMG), for inter-process communication and data flow. The design decisions proposed herein leverage the advantages of existing robotics software architectures and the DDS standards to develop software that is scalable, high-performance, fault tolerant, modular, and readily interoperable with external platforms and software.**

## Nomenclature and Acronyms

| | | |
|---|---|---|
| *ROS* | = | Robot Operating System |
| *DDS* | = | Data Distribution Service |
| *P2P* | = | Peer-to-Peer |
| *OMG* | = | Object Management Group |
| RPC | = | Remote Procedure Call |
| UxS | = | Unmanned x System [x = aerial, underwater, surface, ground, etc.] |

* Software Engineer, Crew Systems & Aviation Operations Branch, MS 492.
† Senior Researcher, Electronic Systems Branch, MS 488, and AIAA Member.
‡ Researcher, Flight Software Systems Branch, MS 492.
§ Researcher, Aeronautics Systems Engineering Branch, MS 130, and AIAA Member.
¶ Researcher, Dynamic Systems & Control Branch, MS 308.
# Researcher, Flight Software Systems Branch, MS 492.
†† Senior Researcher, Crew Systems & Aviation Operations Branch, MS 492, and AIAA Member.
‡‡ Senior Scientist and Head of AI, Crew Systems & Aviation Operations Branch, MS 492, and AIAA Senior Member.

## I. Introduction

TO begin it is important to distinction between an architecture and a framework. In the context of software for robotic systems, architecture is generally used to refer to the structure and layout of the subsystems that exist within it, including the design of the interfaces between subsystems, and the specifications for the underlying mechanisms by which subsystems interact, which may even describe how software should interface with hardware. The specification of the mechanics for interaction typically embody some form of software paradigm, such as describing the system as a collection of unique, loosely coupled services which can be invoked by a client and can invoke other services to achieve some high level goal (Service Oriented Architecture). The architecture does not define the details of how the system should be implemented, such as what language should be used, or what tools or libraries can be used to fulfill the design goals of the architecture. Frameworks, on the other hand, provide all of the details for the implementation. A framework is typically described as a collection of specific technologies, languages, libraries, and tools, which when combined with some development guidelines can be used to create software which effectively achieves the goals of the architectural design. It should be noted that these definitions are not necessarily accepted as global consensus and the two words are often used interchangeably. These definitions are for use in giving context to the following discussion.

## II. Design Approaches

Before designing a software architecture for a robotic system, it is prudent to take a look at how others have done so in the past. What we will find, by looking at existing architectures, is that modern designs tend towards being a hybrid of older architectural designs that leverage the strengths of their predecessors.

### A. Standard Robotic Architectures

*1. Sense Plan Act*

One of the first robotic architectures to emerge was the Sense-Plan-Act (SPA) architecture[1], which came out of the work that Stanford University was doing in the 1960s on the robot Shakey. In the SPA architecture, the general underlying concept is thus: Sensor data which describes the external world should be used to build a world model. On top of that world model, plans can be created to act within the world. Finally, without using the original sensors that were used to create the world model, the robot can execute the planned actions. From the viewpoint of someone interested in designing software to control a robotic arm in a highly static environment, this architecture seems to be simple and sufficient for meeting the designer's needs. However, from the viewpoint of someone wishing to design reactive systems which can function in a dynamic environment, the fact that planned actions are executed in a dead-reckoned fashion, without continuously pulling in updated state information, could be catastrophic. However, the concept of constructing some type of world model through sensor data and planning within that sensed environment is critical to autonomous navigation.

*2. Subsumptive (behavioral)*

As people began to encounter and understand the shortcomings of the SPA architecture, new architectures that relied less on building a complete world model and more on reacting to sensed information began to emerge. In 1986, Rodney Brooks of MIT developed an architecture which would prove to be very influential in the design of reactive architectures. His architecture was called the Subsumption architecture[1], and the main concept that constituted its design was that the robotic system would consist of subsystems which routed sensor inputs through a processor, which consisted of a finite state machine, to the actuators of the robot. This finite state machine effectively instituted a behavior. One important facet of this design is that the system designer could create several behaviors to provide several capabilities to the system, all of which could be executing simultaneously. As an example, consider a robot with a LIDAR scanner weaving its way through obstacles with one behavior while keeping its camera pointed at some specified target using another behavior. These two behaviors could be considered non-conflicting, if the actuators used to achieve the behaviors are different. In the case of conflicting behaviors, such as one behavior which tries to make a robot follow a path and another behavior which executes evasive maneuvers around incoming obstacles, some mechanism is required to govern which behavior has permission to control the actuators. Later, we will see this concept of behavior arbitration frequently. In general, this mechanism could take the form of another finite state machine which has the ability to make decisions using weighted considerations of current sensor inputs, current goals, and potential obstacles to those goals.

*3. Layered approach*

One thing that became apparent over time with the use of behavioral architectures is that, while it was easy to create algorithms and behaviors which enabled robots to react to specific environmental stimuli, it was much more difficult to create behaviors which achieved long-term goals by using simpler behaviors. The problem is that while a simplistic behavior could be created to achieve a goal, such as using wall-following to get from one point in a building to another, it is much harder to optimize such a behavior as the behavior is simply a reaction to a response, without any method of reasoning to make more efficient decisions. Of course, in the arbitration mechanism's finite state machine, one could design some intelligence concerning how to behave under specific circumstances, but when considering the complexity of most environments in which many robots operate, along with the number of behaviors they can express, and the number of possible situations that could occur, the finite state machine tends to grow to unmanageable and unwieldy sizes which can become difficult to design or reason about in their own right.

One solution to these problems was the idea that the two sets of capabilities - intelligent sensing, modeling, and planning from the earlier robotic architectures and the reactive capabilities of the newer architectures - could be combined to create a layered architecture in which modeling, planning, scheduling, and system control are all separate and distinct processing layers. These layers each have their own responsibilities for intelligently making decisions or controlling subsystems, which enables a more efficient method of developing high-level behaviors for robotic systems. The layered (or tiered) approach has since seen many permutations and hybridizations and some form of it constitutes much of the robotic architectures we encounter today.

**B. Subsystem Communication**

The focus of the previous section was on the structural components of the software architecture or, in other words, the methodology by which inputs are transformed into outputs, how the components that affect those transformations are organized, and what exactly they are responsible for. Also important are the underlying paradigms that govern the connectivity of the subsystems within the architecture. These paradigms can largely drive the entire layout of your subsystem modules, by either enabling or limiting the various ways in which they can be connected, modified, or extended. Two communication paradigms for connecting subsystems are *client-server* and *publish-subscribe*.

In a client-server communication pattern, one program communicates to another by means of sending a message and awaiting a response. Typically, these message passing applications, in the context of robotics, are capable of bi-directional message flow, from an ownership of information point of view, so the notion of one program being a server and the other being a client tends to fall out, leaving you with a peer-to-peer communication system.

The publish-subscribe paradigm, on the other hand, describes a communication system in which there are many information topics about which a distributed set of applications express interest or provide information. As an example, consider the scenario where one application is directly connected to a robot's forward facing distance sensor. It publishes values from the sensor to a particular distance sensor topic, where the structure of the sensor data is well established and the name of the topic is unique, such as "ForwardFacingDistanceTopic". Other applications in the system which are interested in that data, such as an obstacle avoidance processing module, can subscribe to the topic and receive updates as they are published by the originating application.

Client-server and peer-to-peer communication styles require the explicit addressing of each participant and manual setup of these end-to-end transports. On the other hand, publish-subscribe systems tend to allow for a scalable, self-discovering network topology based on the publishing and subscribing relationships applications have with the topics that exist within the system. To clarify, in a client-server topology, if you want to send sensor updates to a new client, one must explicitly create a new socket connection with that client and serve the data to them. With publish-subscribe topologies, it is commonly the case that the middleware library or framework which provides publish-subscribe capabilities takes care of the network connections for you, by connecting applications which subscribe to a topic to applications which publish to a topic. One of the major benefits of this feature is that the system is inherently more scalable and the modules in the system less tightly coupled. As applications communicate anonymously (meaning, the module publishing data doesn't need to know anything about the modules subscribing to it), topics and their defined data structures serve as a well-defined interface which can be used in a flexible manner.

Having mentioned the two main types of communication paradigms that employed in robotics software, it is prudent to mention an evolution of the client-server/peer-to-peer paradigm, which is the method of invoking remote procedure calls (RPC), which came about in the standards for CORBA (Common Object Request Broker Architecture) developed by the Object Management Group (OMG) in 1991. The CORBA standard offered a new way for applications to communicate with each other by providing one application to interact with another application by remotely invoking a function in the remote application via a well-defined function interface. RPC via CORBA can be easily understood by drawing a parallel to the semantics of a function call in a language such as C: the function takes N parameters of various type, and returns an output of some type. When communicating with RPC, the client

application typically sends the parameters of some request as function parameters, and the response comes in the form of the return value. The function prototype serves as the common interface between the two applications, similar to how topics serve as the interface between publishers and subscribers. Underneath the hood of whichever vendor implementation of the CORBA standard is being used, the communication is still built upon the mechanics of traditional client-server interactions, which includes the use of explicit endpoint addressing. This is also true of publish-subscribe communication, but publish-subscribe standards and implementations expose the ability to address via multicast, enhancing the performance and ease of use of the software system from a networking perspective. While it is true that there are implementations of CORBA which found ways to utilize multicast addressing, it is generally not the standard method of use and can be, from subjective experiences anyway, somewhat unintuitive to work with.

### C. On Point Designs

One topic that is worth touching on, briefly, is the notion of developing and implementing an architecture that is simply a point design, meaning that it is specifically developed for a single use case and is generally not flexible enough to be effectively used for anything more than its original purpose. An example of this might be a very specific software architecture that drives the operation of a sUAV vehicle that uses very specific set of hardware. Typically, point designs are employed when the developer does not require or foresee the need to add new hardware, expand the software to drive different types of vehicles, or when there is simply not enough time to spend on building a more flexible design. It is true that the development of a point design in robotics typically leads to fast implementation times, lower costs, and better support for a specific use case. However, the goal of the Autonomy Incubator is to develop an architecture and framework that supports collaboration, interoperability, and scalability. Creating a design that assumes specific sensors, vehicle platforms, autopilot boards, or other hardware items would limit the usability of our research only to those that also work under the same hardware and use case assumptions. With that in mind, the following section examines potential methods for achieving the desired usability and flexibility.

### D. Robotics Software Frameworks

The Robot Operating System (ROS) is one of the most popular frameworks in the robotics software design landscape.

#### *1. The Robot Operating System (ROS)*

Despite the name, ROS is not actually an operating system. On the homepage for ROS[2], it accurately describes itself as an "[open source] set of software libraries and tools that help you build robot applications." In the context of earlier discussions, this matches our definition of a software framework. In a summary of what ROS provides to the user, we see that it provides:

- An architecture built around publish-subscribe messaging communication, with RPC capabilities additionally bundled in.
- A set of standardized (within the context of its own ecosystem) message formats common to robotics applications, like distance sensor data, motor commands, and more.
- A format and language for describing the layout and physical properties of robots.
- A tailored build environment, conductive to providing users with a consistent methodology for organizing and building ROS applications.
- A suite of libraries and tools for designing, visualizing, testing, and deploying distributed robotics applications.
- A package management system for installing other existing ROS applications within the ecosystem.
- Client libraries for the C++, Python, and LISP languages on Ubuntu Linux, with experimental support on other Unix-based operating systems.

At a glance, it seems that ROS provides most of the desired features for the design and development of an open software architecture and framework that supports efficient and consistent development within the architecture. However, there are some shortcomings.

- ROS's communication framework is not based upon any open communications standards, so users wanting to integrate their software with ROS have no standard interface for doing so and must roll their own.
- Applications within the ROS ecosystem are not held to any standards by a committee. Because of this, the ecosystem consists of applications with widely varying interfaces and message structures.
- In addition to the above, because there is no oversight, there is no guarantee of quality or correct functionality of any package within the ecosystem.

- In conjunction with the previous problem, because ROS client libraries update in a somewhat frequent manner, and because there is no guarantee that previous and future versions will be compatible, a considerable portion of the application ecosystem is in an isolated, broken, or otherwise unverifiable state of functionality.
- ROS is fairly limited in terms of which platform it is capable of running on.
- Client library support for ARM platforms is somewhat more limited than x86_64 platforms, and the installation process tends to be error prone. This reduces its usability on many of the single board computer devices commonly used in robotics.
- Because effective use of ROS tends to require the use of its provided tools and build environment, there is a significant amount of expertise that a user must gain before being able to work in the ecosystem.
- The build environment relies exclusively on a piece of software called Catkin, which forces the user to adhere to its structure and usage.
- ROS employs the use of a centralized "ROS Core" process, which serves as a single point of failure in the network topology.

With both the advantages and disadvantages in mind, we believe that while many of the goals and characteristics of ROS are in line with our visions of an open robotics architecture and framework, the disadvantages present several challenges that stand in the way of interoperability and allowing for the conjunction of systems that exist in ecosystems outside of ROS. To address these problems, we turn to the Open Middleware Group's (OMG) Data Distribution Service (DDS) for Real-Time Systems[3].

*2. Data Distribution Service (DDS)*

DDS is a standard for publish-subscribe middleware which brings several new features, abstractions, and guarantees to the table in the design of distributed software systems and systems of systems. As DDS is a standard, there are many vendors which have implemented middleware suites which fulfill the requirements of the standard and provide other extended functionality and tooling on an individual basis. All of the implementations of DDS middleware are guaranteed to be interoperable at the wire-protocol level, so there is generally no risk of vendor lock-in when choosing an implementation. Communication using DDS centers around the interactions between a collection of DDS defined entities. As shown in Figure 1, these entities are Topics, Participants, Publishers, Subscribers, Datareaders, and Datawriters. At a basic level, a Participant has both a Publisher and a Subscriber, and the Publisher and Subscriber contain various datawriters and datareaders, respectively, which communicate with each other about specific Topics, which correspond to messages about a certain type of data.

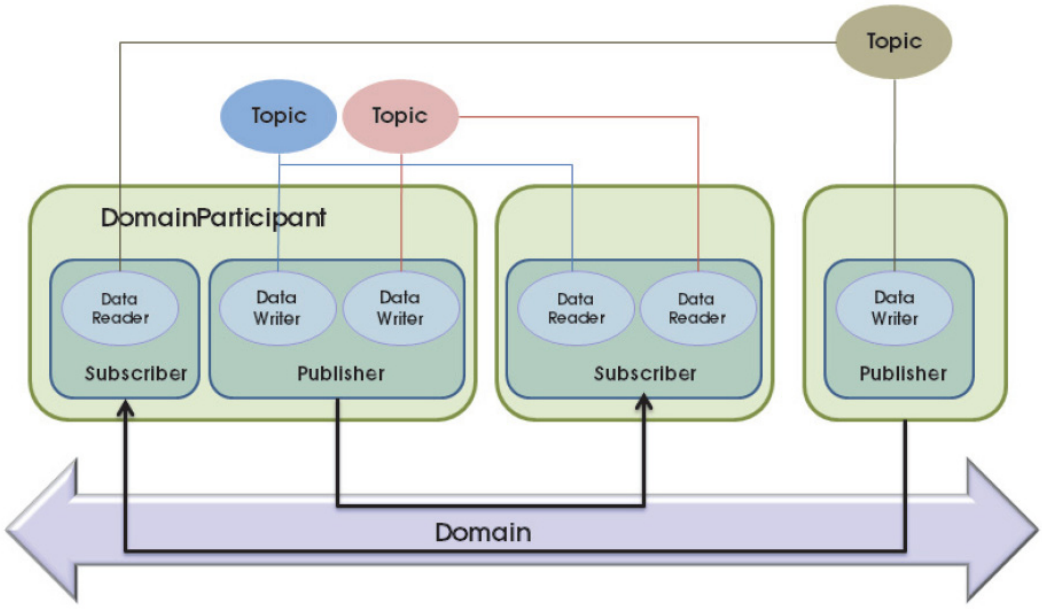


*Figure 1: Publish-Subscribe Middleware Model [10]*

Some of the features of DDS that make it stand out for our purposes:

- A completely decentralized network topology; there is no centralized message broker. All applications discover and communicate with each other directly.
- DDS network topologies are completely "plug-and-play". Discovery is a persistent function carried out by the middleware, allowing asynchronous joining and leaving of participants.
- Various measures of Quality of Service (QoS) which are associated with each type of entity. These QoS measures provide certain guarantees which are fulfilled by the middleware concerning transactions between the various entities. Examples include reliability, durability, deadline, transport mechanism, history, and more.
- Support for Windows, Linux, and Mac, and also actual real-time operating systems such as VxWorks.
- Support for more languages, including C, C++, C#, Java. Additional language support is also provided by certain vendors, such as for ADA, Scala, and Javascript languages.
- Lightweight, small footprint implementations for embedded systems.
- Portability. It can be used in existing frameworks and build environments with few modifications.
- Communication over heterogeneous network technologies, such as ethernet, wifi, bluetooth, serial, RF, etc.
- OMG's IDL specification for files expose the data types referred to by topics. If a system exposes its data model and its IDL files, it is immediately possible to achieve interoperability with other DDS enabled software systems.
- Several higher level communication capabilities that are typically reinvented by teams that build distributed systems, such as filtering of data, reliability guarantees, liveliness status management, wire-level failure handling.
- Built in capabilities for building highly reliable, redundant systems with deterministic networking configurations.
- Communication framework abstracted away from the users of a DDS-based framework. All these need to know is how to create readers, writers, topics, and data types, all of which are greatly facilitated and simplified by the middleware and its tooling.

## III. The Aeon Architecture

The Aeon Architecture is designed to achieve the following high-level goals of the LaRC Autonomy Incubator:

- Rapid integration of both in-house and third party software modules
- Platform agnosticism
- A system that is high performance, fault tolerant, and redundant
- A high level of modularity with extremely loose coupling to achieve reusability and flexibility
- Abstraction of networking details and boilerplate from researchers and third parties
- Distributed communications that enable software to run anywhere
- Interoperability via well-defined communication interfaces based on standards

Our proposed solution comes in the form of what could be considered a lightweight alternative to ROS that uses RTI™ Connext DDS middleware implementation as its communication layer. What is meant by lightweight is that, for the most part, our framework does not impose any build environment requirements or restrictions. The core unit of our software framework is an application called AeonCore. This application is composed of a few important subsystems:

- AeonDDS – A project that is shared by all application nodes in the system and contains the IDL, Application, and QoS XML files used by the DDS middleware. By every application having access to this, the developer can easily add new interfaces and find existing interfaces when developing a new node.
- Base Application Class – A class that sets up the application in a ready-to-use manner. The developer need only derive their own application from this class and implement their own specific application logic.
- DDS Manager Class – This class facilitates easy construction of all DDS entities in the middleware layer of the application without the developer needing to know how to manually use the DDS API to manually accomplish these tasks. It also acts as an easy to use interface to the DDS entities for when the user wants to read or write data.

By utilizing RTI's XML-based DDS application creation functionality, the setup of DDS-enabled applications is greatly simplified, and much of the knowledge required to create and manage DDS entities is abstracted away from the user. Combined with the fact that all communication related interfaces and settings are stored in a shared set of XML and IDL files, it is easy for any developer in the system to create or modify interfaces between application nodes without needing to be an expert in the DDS middleware's API. Because creating and modifying interfaces is as easy as modifying IDL and XML files, and since the user can simply modify a couple of functions in the AeonCore skeleton application, our goal of enabling rapid integration of in-house and third party software modules is achieved. The typical workflow involved in creating a new node is as follows:

- Choose and include the IDL files that contain types you want to read or write
- Create an application profile that sets up the various DDS entities you need in your application in the app profile XML file, choosing the various QoS settings that make sense for your node.
- Modify the DDS Manager class to load your app profile.
- Create a class that inherits from the base application class and implement your logic as you see fit
- Use the DDS manager as an interface to the DDS datareaders and datawriters
- Optionally use the DDS WaitSet object in the application class to handle new samples, status updates, etc.

Beyond core node, we examine the overall structure for the onboard vehicle software architecture. Similar to the multi-layered approach that we examined earlier, our architecture expresses itself as a hybrid architecture pulling in elements from SPA and Subsumption. As an overview, the architecture is split into four layers:

- Data Production
- Executive
- High-Level Behavioral
- Service.

The first layer is the *Data Production Layer*. The Data Production layer functions as a funnel that pulls in processed sensor data from several sensor processing nodes that handle outputs from real sensor hardware or the outputs of other sensor processing nodes. Eventually, once the sensor data is as processed as it can be, the second highest tier of processing nodes format the data into a standard data structure known as an Entity State. The Entity State structure contains position, velocity, and acceleration values and variances for both the vehicle itself and other entities in the world. Several Entity State messages are published by a varying array of sensor processor nodes and are fused together using an asynchronous filter in both a SelfEstimator and WorldEstimator node. These two nodes provide a final, fused, best-effort Entity State structure which can be used anywhere in the rest of the system where self state or the state of external objects is required. As an example, the vehicle controller which requires knowledge of the vehicle's state, subscribes to the SelfEstimator's output, and the PathPlanner module, which requires knowledge of obstacles, subscribes to the WorldEstimator's output. In addition to the condensed, standardized format of Entity State, the sensor processor modules can also publish their specific data streams, such as an optical flow sensor module publishing data specific to optical flow measurement parameters. In this way, all data produced by any sensor can be made available to the rest of the system asynchronously and on demand. By using various QoS settings, even more complex and powerful behaviors can be achieved in the Data Production layer.

The next layer, which is currently under development, is the *Executive Layer*. The Executive Layer contains modules which are responsible for handling the permissions of other nodes within the system, making critical operation decisions, and handling overall system status and capability degradation.

Following the Executive Layer is the *High-Level Behavior Layer* (HLBL). The HLBL contains nodes which chain together nodes in the final layer, the *Service Layer*, to perform more complex behaviors than any single service on its own could achieve. Thinking about the subsumption architecture and SPA architecture from before, services can be thought of as behaviors that take inputs from HLBL nodes, other Service Nodes, and Data Production Layer nodes to actuate the vehicle's hardware. The HLBL can use the various discovery and QoS properties of the DDS Middleware to dynamically chain services together to execute complex tasks, such as navigation to specific points, searching for targets, or controlling other hardware to interact with the environment in different ways. Some services have capabilities that do not involve actuating hardware directly, such as path planning services. These services typically handle the “Plan” part of the SPA paradigm by taking the outputs from the Data Production Layer (“Sense”), processing those outputs, and feeding the inputs to services which can “Act”. In this manner, we have designed a system which reaps the benefits of being able to asynchronously sense, plan, and act, while instilling the planning and acting capabilities in a set of reusable behaviors known as services, and being able to schedule those services using HLBL nodes. See Figure 2 for a high level view of the architecture with example nodes provided in each layer.

Another aspect of the architecture which helps achieve the Autonomy Incubator goal of a distributed robotics system is the fact that each vehicle has its own DDS partition within the Vehicle DDS domain. This keeps all of the internal messaging of the vehicle contained to its own local, onboard network. There exists in each vehicle software set a node that acts as a bridge to the GCS domain and also to other vehicle bridge nodes to enable distributed communication between both peers and applications in the ground control domain.

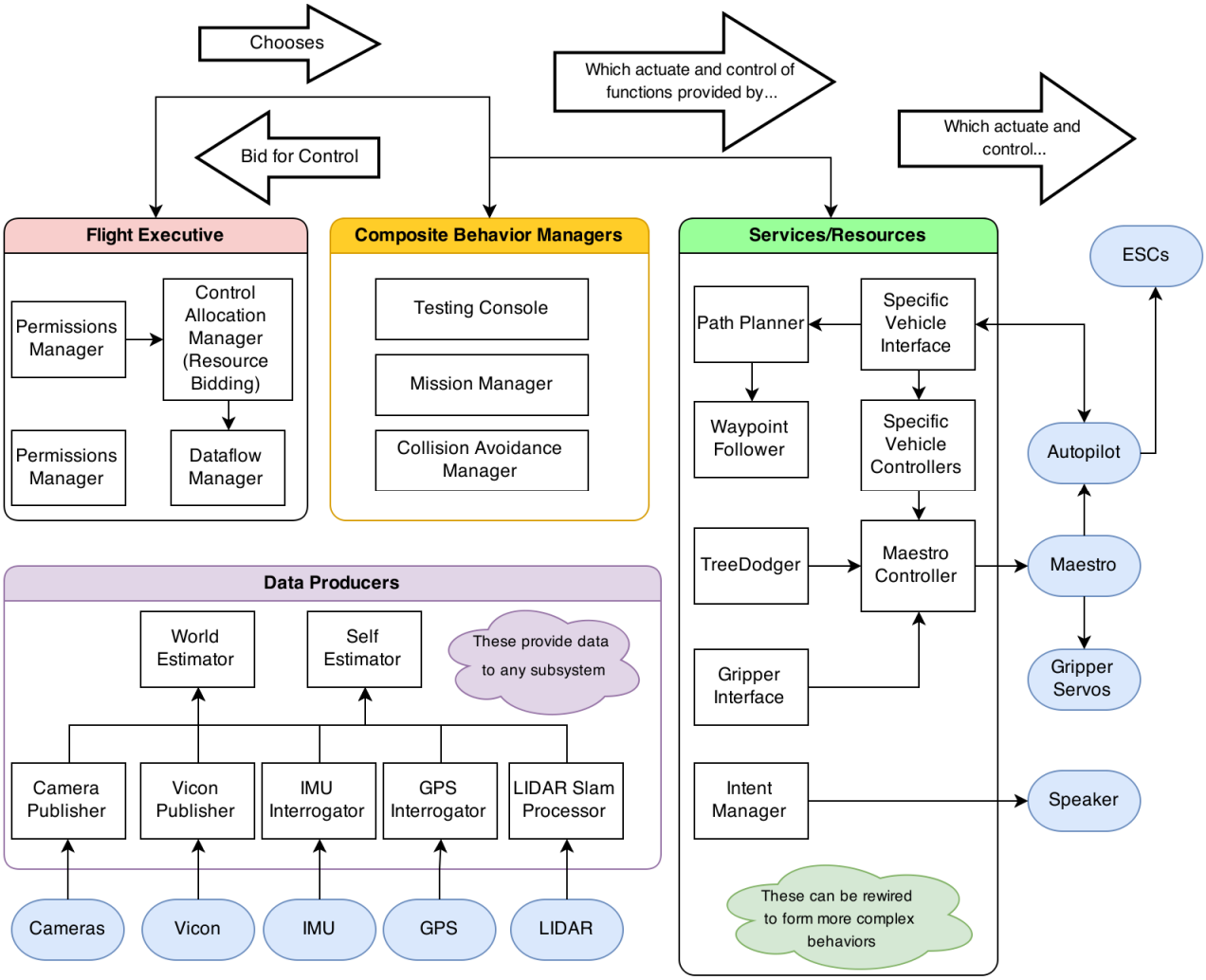


*Figure 2: AEON Architecture*

## IV. Future Work

Moving forward, our next steps involve building additional capabilities onto the software ecosystem that executed the previous demo. We have already been able to run most of the existing software on new vehicle platforms that can carry more sensors and heavier payloads, which will be required to perform more complex missions. By rapidly scaling the software ecosystem node by node, with each researcher working on the pieces in which they have expertise, we hope to demonstrate the significant strength and flexibility of our architecture and approach to solving distributed robotics. Some specific areas that will see significant improvement in the immediate future are the development of the more advanced, asynchronous input, SelfEstimator and WorldEstimator filter modules, the development of more robust control algorithms in the VehicleController node, the development of sensing and avoidance algorithms that will feed vehicle navigation behaviors, and also the development of ground control modules that will provide novel methods of human interaction with the vehicles. Having finished and validated the framework at a prototype level, we

are confident that the system will continue to improve and further achieve the goals laid out for the LaRC Autonomy Incubator.

## V. Conclusions

The Autonomy Incubator employs an Agile development methodology and so the software system is rapidly evolving in support of monthly showcases. In one of our most recent demonstrations, we utilized the framework and system to enable a quadrotor sUAV to autonomously navigate to waypoints while dynamically replanning around dynamic obstacles. As an example HLBL node, we used a simple console that allowed the operator to request the vehicle to navigate to particular waypoints. These requests were passed to a PathPlanner service node, which took inputs from the SelfEstimator and WorldEstimator nodes to plan a safe route around any detected obstacles. Figure 3 shows the relationships and interactions between the nodes in this demonstration.

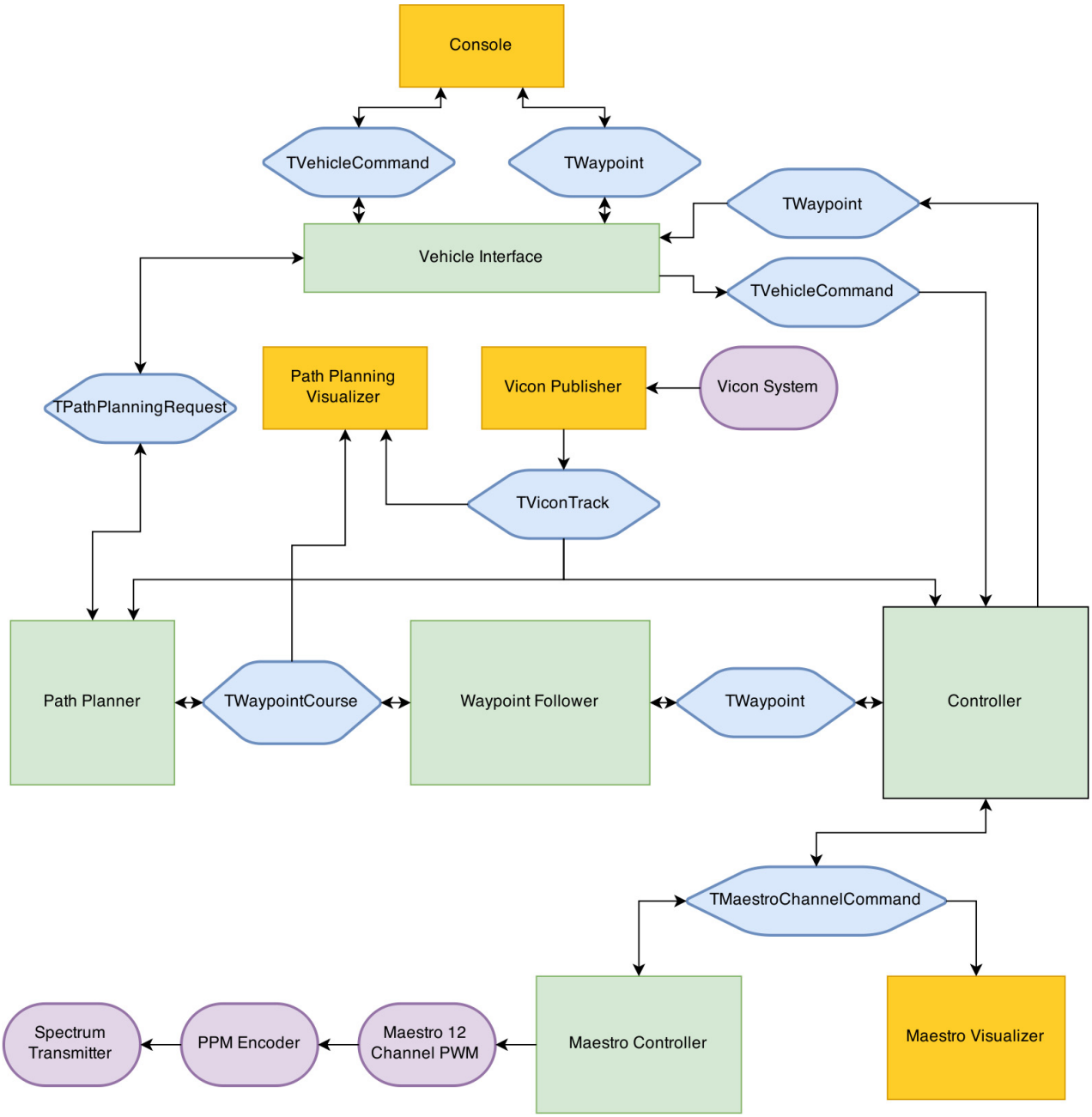


*Figure 3: AEON architecture and framework utilized in recent Autonomy Incubator demonstration*

In developing the software for this demo, the AI[4-9] accomplished many of the goals for our architecture, particularly by developing the AeonCore node which served as a framework on which several researchers could quickly implement functionality based on their individual research paths in application nodes that were distributed across a number of platforms and languages. With regards to our goals in developing a distributed ground control software ecosystem, a few prototype applications were developed that allowed capabilities such as voice command of a vehicle and also the visualization of data coming off of the vehicle for data gathering missions involving vehicles carrying sensor payloads.

## Acknowledgments

This effort would not be possible without the ongoing effort of all members of the Autonomy Incubator team at NASA LaRC. All work could not be possible without the drive, energy, and dedication of the AI members, peers, champions, and families. NASA student intern efforts during the spring of 2015 have also aided this work. Special thanks go to Gil Montague, Matt Mahlin, Irvin Cardenas, Jacob Beck, and Sarah Voorhies.

## References

[1] Kortenkamp, D., Simmons, R. “Handbook of Robotics”, Chapter 8: Robotic Systems Architectures and Programming

[2] Woodall, W. “ROS on DDS,” http://design.ros2.org/articles/ros_on_dds.html

[3] Open Middleware Group (OMG) Data-Distribution Service (DDS) for Real-Time Systems, http://portals.omg.org/dds/.

[4] L. Tran, C. D. Cross, N. H. J. Motter, M. A, G. D. Qualls, P. M. Rothhaar, A. C. Trujillo and B. D. Allen, "Reinforcement Learning with Autonomous Small Aerial Vehicles in Cluttered Environments," in *Aviation 2015*, Dallas, TX, 2015.

[5] P. M. Rothhaar, C. D. Cross, N. H. J. Motter, M. A, G. D. Qualls, L. Tran, A. C. Trujillo and B. D. Allen, "A Flexible Flight Control System for Rapid GNC and A Flexible Flight Control System for Rapid GNC and," in *Aviation 2014*, Dallas, TX, 2015.

[6] H. J. Neilan, C. D. Cross, A. M. Motter, P. M. Rothhaar, G. D. Qualls, L. Tran, A. C. Trujillo and B. D. Allen, "Using Multimodal Input for Autonomous Decision Making," in *Aviation 2015*, Dallas, TX, 2015.

[7] A. C. Trujillo, C. D. Cross, M. A. Motter, H. J. Neilan, G. D. Qualls, P. M. Rothhaar, L. Tran and B. D. Allen, "Collaborating with Autonomous Agents," in *Aviation 2015*, Dallas, TX, 2015.

[8] G. D. Qualls, C. D. Cross, M. A. Motter, H. J. Neilan, P. M. Rothhaar, L. Tran, A. C. Trujillo and B. D. Allen, "Operating in “Strange New Worlds” and Measuring Success – Test and Evaluation in Complex Environments," in *Avation 2015*, Dallas, TX, 2015.

[9] M. A. Motter, C. D. Cross, H. J. Neilan, G. D. Qualls, P. M. Rothhaar, L. Tran, A. C. Trujillo and B. D. Allen, " Deciding to Go Around via Machine Learning," in *Aviation 2015*, Dallas, TX, 2015.

[10] RTI Connext DDS Core Libraries and Utilities Getting Started Guide, Version 5.1.0., https://community.rti.com/rti-doc/510/ndds.5.1.0/doc/pdf/RTI_CoreLibrariesAndUtilities_GettingStarted.pdf