Toward standard interfaces for high-level Autonomy Simulation with MOOS-IvP

Mathew Schwartzman
Ocean Engineering Lab
NEREID Ocean Conservation, Inc.
Bedford, MA, USA
mcschwartzman@outlook.com

***Abstract*—Digital simulation of robotic systems is a near-universal need in the development of new autonomy. That need is exponentially increased in the case of oceanographic engineering, which remains one of the most expensive applications of the technology. While many modern robotics simulators aim to implement high-fidelity physics engines and 3D graphics, this paper highlights the conversely lightweight toolset used in the MOOS-IvP ecosystem for high-level and multi-vehicle simulation. Additionally we evaluate a modular container stack, WebMOOS, containing an API and other microservices that serve a set of standard interfaces for new engineers and scientists.**

Keywords—autonomy, robotics, maritime, api, docker, ROS, MOOS

I Introduction

In the field of Marine Robotics, precious sea time is an expensive commodity valued by engineers and mariners alike, but rarely yielded from important maritime operations. This scarcity has fortunately resulted in numerous innovations in the field of marine simulation, with higher fidelity simulation environments being developed all the time. Gazebo, an open-source standard originally developed alongside the Robot Operating System (ROS and ROS2) middleware, has grown more ubiquitous among these environments, using GPU acceleration to render realistic underwater and surface 3D environments [1]. These simulation environments are excellent for testing low-level code with high-fidelity, but may not scale well when testing more high-level and multivehicle operations. For these scenarios, a thinner simulator is more practical. Sometimes used as a backseat autonomy controller for ROS-enabled vehicles, MOOS-IvP and its lightweight simulator uSimMarine can fulfill this role [2]. This paper presents WebMOOS, a standard collection of interfaces wrapped around the MOOS-IvP toolset to empower high-level autonomy engineers to more rapidly develop.

II Background on MOOS-IvP

MOOS (Mission Oriented Operating Suite) is a feature-complete middleware developed in 2001 by Paul Newman of Oxford university [3]. The subsequent extension MOOS-IvP (Interval Programming), maintained by Michael Benjamin at MIT, introduced a behavior-based interval programming architecture to handle multi-variable optimization problems common to modern robotics development. This high-level autonomy system is used by the MIT Marine Autonomy Lab and the MIT Naval Architecture students, as well as several labs and startups in the marine robotics space. The MOOS-IvP libraries encompass many typical scopes of autonomy design, providing internal standards and tools for definition of vehicle dynamics, inter-vehicle communication, and simple environment simulation.

MOOS itself can be thought of as a lightweight alternative to ROS, LCM (Lightweight Communications and Marshalling), and other robotics middlewares, handling interprocess communication with a pubsub (publish and subscribe) architecture [4]. MOOS forgoes any sort of dedicated external message definition, instead passing messages between MOOS “apps” as strings and floats. In a typical MOOS architecture, a dedicated app might serve as an interface to external instruments like radar, or perhaps to other apps, passing executive commands or sensor data. The transport of these messages is itself managed by another MOOS app, called the MOOSDB. In its most base form, the MOOSDB simply serves as a storage container for these messages, or “MOOSvars”. Other apps can subscribe to these MOOSvars to receive updates to their values, and can similarly publish to MOOSvars as well.

MOOS-IvP adds enhancements like the Helm IvP MOOS app, which uses a behavioral approach to optimize a speed and heading solution around multi-variable problems like collision avoidance and navigation [5]. For example, while the vessel is transiting along a path of waypoints, radar data about an incoming contact can trigger the spawning of a Collision Avoidance (COLREGs) [6] behavior that outputs a desired speed and heading along with the original waypoint transit behavior. The output of both behaviors are weighted and contribute to the final speed and heading decision of the autonomy system.

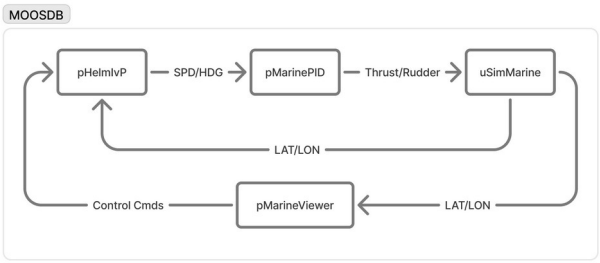

Figure 1: A simplified diagram of a typical MOOS-IvP-enabled simulation

MOOS-IvP also adds the utility uSimMarine, a simple, lightweight simulation environment that allows configuration of basic environmental dynamics like buoyancy and current and subscribes to thruster values and rudder angle [2]. The simulator then publishes telemetry data over time to the MOOSDB, where the Helm uses the new data to make decisions about speed and heading in the next iteration.

III Standardization with WebMOOS

While MOOS-IvP is feature-complete and serves as an excellent standalone option for autonomy control, its lightweight, no-frills nature necessitates a simple custom interface, sharing floats and strings over the MOOSDB client protocol. And while intersystem marshalling protocols have been developed to harden the MOOS configuration and messaging architecture [7], such enhancements requires additional frontloaded technical investment that can add further barrier to entry. This paper presents a repository packaging a few thin architectual additions that allow scientists and new engineers to more easily integrate with modern tools.

III.A Southbound Communication

Southbound communication is defined as messaging from a higher-level control system to a lower-level control system. In this case, the lower-level systems are those closer to the default autonomy control system, MOOS-IvP. An example would be a user issuing a "deploy" command to the vehicle, which would then begin a planned mission, perhaps to survey a location of interest. MOOS-IvP offers command-line tools to manually "poke" commands like this to the MOOSDB, similar to the rostopic pub command, but in the efforts of adopting a more commonly used standard, a REST (Representational State Transfer) API (Application Programming Interface) was developed, allowing for simple http POST requests.

With the most popular programming languages of the past 10 years being Javascript and Python, [8] and the most popular programming technologies being web frameworks like Node.js and React.js, it's likely that web developers make up the majority of software engineers globally. And with the advent of generative Large Language Models (LLMs) like ChatGPT, it's easier than ever to learn and strap basic full-stack web applications. The REST API presented here utilizes the python-moos library [9] and the FastAPI framework to provide a simple web API written in python.

The most important endpoint of the API is the "post-moosvar" route. With python-moos encapsulating the API into a dedicated MOOS app, the FastAPI library makes it trivial to define REST endpoints to allow interaction between higher-level components and the autonomy system. The format of Southbound communication was decided to be Javascript Object Notation (JSON), a well-known modern standard for web communication.

The "post-moosvar" endpoint allows for the posting of any MOOSvar from a REST client to the MOOSDB but the primary use-case would be posting of aforementioned mission commands like "deploy mission" or "return to home". Indeed, a standard included in nearly every moos-ivp sample mission is a mode tree that checks the value of the MOOSvars "DEPLOY" and "RETURN" to command the vehicle accordingly. Though these postings are typically singular, the POST endpoint can be utilized for rapidly repeated synchronous calls to the MOOSDB assuming the underlying FastAPI architecture can support them.

While in theory it's possible to also expose a GET endpoint that allows for a direct http call to the value of any specific MOOSvar, in practice most interactions with these variables necessitate a more asynchronous and continuous approach. In the first iteration of WebMOOS, we subscribe to vehicle telemetry messages. For this purpose, an alternate northbound interface was architected.

III.B Northbound Communication

Northbound communication is defined as messaging from a lower-level component (like the autonomy control system) to a higher-level component, which here could be any microservice closer to the user, e.g. a web-based UI (user interface), like the one included in the linked repository. It's important to note that the concepts of northbound and southbound messaging are relative, and indeed, a high-level autonomy system like MOOS-IvP will be further "north" than, for example, vehicle motion controllers and sensor drivers.

The pubsub methodology is not unique to robotics middlewares like MOOS and ROS. It's an architecture used to provide real-time updates in large-scale social media websites and IOT (internet of things) networks [10]. An OASIS standard protocol [11], MQTT (Message Queuing Telemetry Transport) was a natural fit for WebMOOS to provide asynchronous messaging for northbound external clients of the MOOSDB.

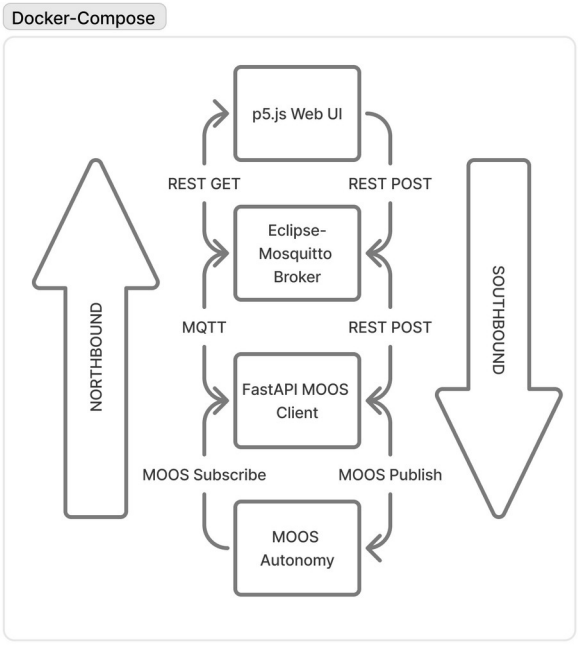

Figure 2: Dataflow is bidirectional, and the flexibility of the FastAPI app allows for multiple data transports

In the first iteration of WebMOOS, limited messages are published to the MQTT broker from the FastAPI MOOS app. The most important, NODE_REPORTs, provide telemetry and autonomy mode information about nearby vehicles or "nodes". This data proved sufficient for evaluating WebMOOS, and adding MOOSvars to the MQTT simply requires a call to single-publish the variable value on a desired topic.

The MQTT library used by the API also required a dedicated message broker, Eclipse-Mosquitto [12] to serve data between subscribed clients. The API MOOS client first subscribes to the NODE_REPORT MOOSvar using the custom MOOSDB protocol, then parses the data into JSON before publishing it to the "/node-report" Mosquitto topic. While not strictly required by the broker, JSON was used for consistency with the southbound REST interface. The Mosquitto broker, and all other microservices, are defined as containers in a docker-compose.yml file, and components

can be disabled as desired, if wishing to run any or all portions on “bare metal”, i.e. without containerization.

III.C A bare-bones Web UI

With northbound and southbound interfaces established, a lightweight Web-based GUI (Graphical User Interface) was developed to showcase WeMOOS’s ease-of-use. As with core MOOS-IvP, the intention here was not to build a high-fidelity graphical/physical simulation, and was instead to augment the autonomy system with high-level interfaces.

That said, in evaluating a simple Web UI technology, it was prudent to select something with a low barrier to entry to scientists and new engineers. While larger frameworks like React.js or GeoDjango would likely serve as more robust alternatives and development of such tools is encouraged by the author, the Javascript graphics library p5.js was selected for its ease of implementation [14].

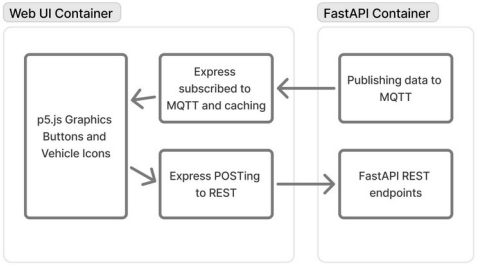

Figure 3: The simple p5.js Web UI, a stateless UI that simply draws data as it’s received from the API

A simple express.js server was spun up to serve the p5.js library and sketch code statically. The p5 architecture provides easy-to-use convenience functions: “setup”, which gets run once on page refresh, and “draw” which gets repeatedly called each frame. Whenever a frame is drawn, the frontend makes a call to the express server, which has subscribed to the node-report MQTT topic and cached data about the nearby vehicles. The UI then simply paints points on the screen for each vehicle in the cache. Because it’s simply reading data from a short-term cache and painting it, the UI is completely stateless and there’s no risk of becoming “out of sync” with the express server, the API, or the underlying MOOS-based autonomy.

Similarly, the Deploy and Return buttons on the UI are simply mapped to POST requests to the express.js server, which then makes direct requests to the FastAPI container, to update MOOSDB variables directly. In WebMOOS, the default mission of 2 vehicles transiting to a waypoint mission begins with the click of the Deploy button and pressing the Return button commands the vehicles to return to their respective starting points.

IV Conclusion and Future Work

This paper describes the first iteration of WebMOOS and the design tenets of its architecture. While the tenets of modularity, extendability, and lightweight implementation are unlikely to change, additional functionality and improvements are pending. The contributions of this research are an initial microservice architecture on which to iterate. WebMOOS is in a state of early evaluation and development, and is slated for beta-testing at the NEREID Ocean Conservation Hackathon in June of 2025.

Along with an improved and more feature-complete (but still lightweight and stateless) Web UI, future developments will include a more practical MQTT server, allowing API clients to dynamically register for larger subsets of the MOOSDB. This will be a requirement for building a more robust and practical API, allowing users to develop more complex applications outside of the MOOSDB.

The described architecture is meant to eliminate or reduce barries to entry in the fields of robotics and particularly high-level autonomy development. It is the intention of the author to work with autonomy developers and scientists particularly interested in multi-vehicle autonomy to evaluate the effectiveness of the architecture in testing as well as education. The last core tenet of WebMOOS is ease-of-use, and the true test of the toolset will be its utility in the various simulation environments where ubiquity and modularity are more important than fidelity.

Acknowledgment

The WebMOOS stack is an open-source project that welcomes community contributions. All development contributions are publicly recorded on the github repository at https://github.com/mcschwartzman/webmoos/tree/main. I’d like to thank Mike Benjamin at the MIT Marine Autonomy Lab for his advice in early MOOS-IvP development and his consistent updates to the core MOOS-IvP repositories.

References

1 *B. Bingham et al., "Toward Maritime Robotic Simulation in Gazebo," OCEANS 2019 MTS/IEEE SEATTLE, Seattle, WA, USA, 2019, pp. 1-10, doi: 10.23919/OCEANS40490.2019.8962724.*

2 M. Benjamin, “uSimMarine: Basic vehicle simulation,” 2025 [Online]. Avail. https://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=IvPTools.USimMarine

3 M. Benjamin, “Overview of the MOOS-IvP Autonomy Project,” 2025 [Online]. Avail. https://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=Helm.MOOSIvPIntroduction

4 A. S. Huang, E. Olson and D. C. Moore, "LCM: Lightweight Communications and Marshalling," 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, Taiwan, 2010, pp. 4057-4062, doi: 10.1109/IROS.2010.5649358.

5 M. Benjamin, R., John J. Leonard, Henrik Schmidt, and Paul M. Newman. "An overview of moos-ivp and a brief users guide to the ivp helm autonomy software." (2009).

6 International Maritime Organization. "Convention on the International Regulations for Preventing Collisions at Sea, 1972 (COLREGs)." (1972).

7 T. Shneider "Goby3: A new open-source middleware for nested communication on autonomous marine vehicles." In *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pp. 236-240. IEEE, 2016.

8 StackOverflow, “2024 Developer Survey”, survey.stackoverflow.co, https://survey.stackoverflow.co/2024/technology, (accessed March 31 2025)

9 M. Saad Ibn Seddik, “Python-MOOS”, github.com https://github.com/msis/python-moos (accessed March 31 2025)

10 Google Cloud “Architectural Overview of Pub/Sub” google.com https://cloud.google.com/pubsub/architecture (accessed March 31 2025)

11 Silva, Daniel, Liliana I. Carvalho, José Soares, and Rute C. Sofia. "A performance analysis of internet of things networking protocols: Evaluating MQTT, CoAP, OPC UA." *Applied Sciences* 11, no. 11 (2021): 4879.

12 Mosquitto, Eclipse. "Eclipse mosquitto." *Eclipse Mosquitto* (2018).