Designing an Operator Control Unit for Cooperative Autonomous Unmanned Systems

Paul Candela, Alexander Xydes, and Adam Nans

Space and Naval Warfare Systems Center Pacific, San Diego, CA 92152

ABSTRACT

Interfacing with and supervising a team of heterogeneous, modular robotic systems across various, dynamic mission sets poses significant demands on the design of an operator control unit (OCU). This paper presents an architecture and OCU design which focuses on three key elements addressing this problem: 1. A well-defined system architecture for external interfacing, 2. Dynamic device discovery to advertise current capabilities, and 3. A flexible front-end to convey situational awareness.

The system architecture and external interface is influenced by work done in 4D/RCS (Four Dimensional Real-time Control System) to provide a hierarchical decomposition of the system autonomy. The platform independent, web-based OCU is capable of interacting with any level of the system-of-systems from low-level teleoperation of a single vehicle to section-level concerns such as commanding coordinated tactical behaviors across multiple vehicles. The architecture design facilitates dynamic discovery of the team of vehicles and their respective payloads. The OCU subscribes to pertinent data from these advertised systems, providing multiple human supervisors the ability to interact with and receive concurrent visual representation of them and their status. System failures, payload detections, and user decision prompts are displayed in a prominent yet unobtrusive manner.

The OCU was used to control a heterogeneous team of four vehicles performing coordinated route clearance operations in a set of four demonstration vignettes. Results of a technical exercise using this system are described herein.

Keywords: autonomous vehicles, ROS, MOCU, operator control unit

1. INTRODUCTION

Carrying out dynamic mission sets across a cooperative team of heterogeneous, modular robotic systems poses challenges for remote supervision. This paper describes three elements key to providing that capability. These elements were designed and used during a technical exercise conducted with four heterogeneous vehicles performing coordinated route clearance operations. A system architecture with a well-defined external interface provides a solid foundation and simplifies access to the system-of-systems from low-level teleoperation of a single vehicle to section-level concerns such as commanding coordinated tactical behaviors across multiple vehicles. This architecture facilitates dynamic discovery of the team of vehicles by advertising their respective capabilities and payloads. Finally, subscribing to these discovered vehicles is a modular operator control unit (OCU) that conveys situational awareness and utilizes the architecture interface to provide command of the system.

2. SYSTEM ARCHITECTURE

A well-defined system architecture provides the bedrock upon which control of a team of autonomous systems can be built. This system architecture is influenced by work done in Four Dimensional Real-time Control System (4D/RCS)[1] to provide a hierarchical decomposition of the system autonomy. Using a hierarchical architecture provides access to both low and high-level control methodologies which require different levels of engagement from operators. This provides operators the ability to choose how much effort to expend controlling the systems. Low-level control, such as teleoperation, requires a high level of focus and engagement but also provides very

Further author information:
{paul.candela, alexander.xydes, adam.nans}@navy.mil

Unmanned Systems Technology XIX, edited by Robert E. Karlsen, Douglas W. Gage, Charles M. Shoemaker, Hoa G. Nguyen, Proceedings of SPIE Vol. 10195, 101950L · © 2017 SPIE
CCC code: 0277-786X/17/$18 · doi: 10.1117/12.2267163

fine-grained control of an individual system. Alternatively, high-level control, such as a section-level controller, requires less focus and engagement of each individual system — instead providing coarse-grained control of the entire section of systems. Such control might consist of choosing a single behavior like route clearance or bridge crossing or alternatively, formulating a sequence of behaviors for the section to complete. In addition, formulating a sequence of behaviors can be accomplished prior to mission execution. This reduces the cognitive load required by increasing the amount of time available to complete the formulation.

The interfaces between the levels of control are also well-defined by the system architecture. The means of communication, messaging, and forms between system levels are defined as messages, services and actions in the Robot Operating System (ROS)[2] framework. Moreover, vehicle level planners are required to conform to a ROS pluginlib[3] C++ API, allowing them to be built separately from the system software and loaded at runtime. This reduces or possibly eliminates the amount of changes required to add a new planner to the system. This system architecture also provides arbitration between operator and higher-level planner access at each level. This prevents an operator commanded behavior currently executing from being overridden by a higher-level planner, which could take the operator by surprise and reduce their trust in the system.

This system architecture is broken down into five levels for the mobility stack: section (behavior sequencing), vehicle (behavior coordination), maneuver, reactive, and drive-by-wire (DBW). These levels evolved from the levels detailed in 4D/RCS:[1] Section, Vehicle, Subsystem, Primitive. Each level handles a different timeframe and planning horizon while also replanning at different rates. For instance, the lowest levels (drive-by-wire) output on the order of 5-10 milliseconds, and the vehicle level (a higher level) replans on the order of every 5-10 seconds. Viewed in terms of planning timeframe, the lowest levels (drive-by-wire) construct plans for the next 50 milliseconds, whereas the vehicle level plans for either part of a behavior or the full behavior at one time. Not all of the levels currently contain multiple planners. Figure 1 illustrates the current system architecture.

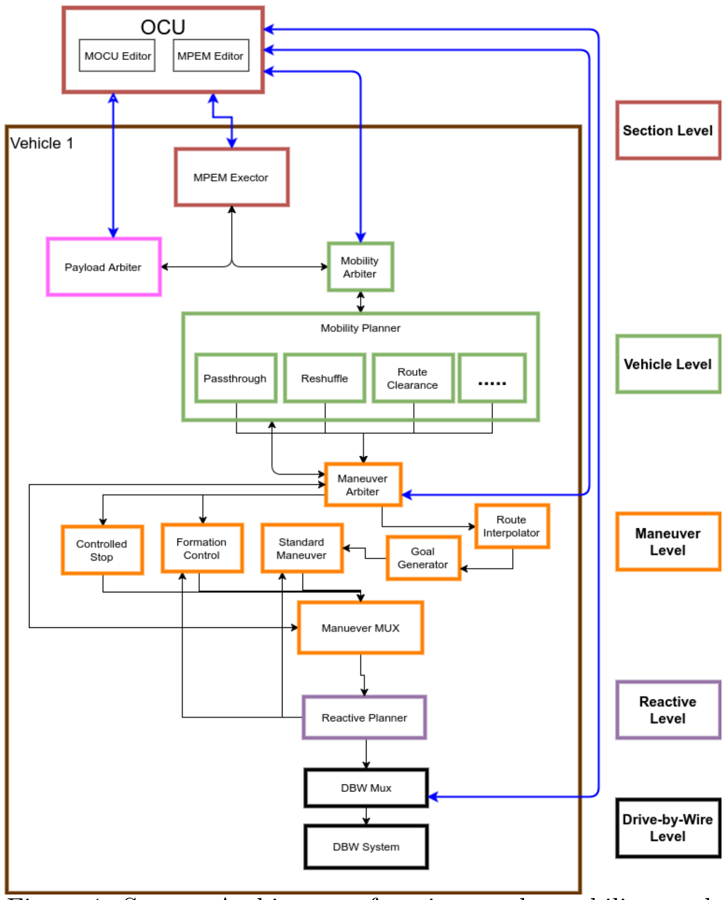


Figure 1: System Architecture focusing on the mobility stack

At the section level only one planner currently exists: the Mission modeling, planning, and execution module (M2PEM).[4] This provides the operator the ability to setup a hierarchical sequence of behaviors for the entire section of systems. These can range from a flat sequence of behaviors, to multiple levels of parallel sequences.

At the vehicle level multiple planners exist, including a single pass-through planner that forwards the desired plan from the section level down to the maneuver level. Other planners provide the ability for the team to cross bridges in a coordinated fashion, perform route clearance, maintain a line-haul formation (also known as convoying), and reposition the team of systems in a new order.

The maneuver level provides arbitrated access to three main planners: a single-vehicle maneuver planner, a multi-vehicle formation planner and a single-vehicle controlled stop planner. The single-vehicle planner provides single-vehicle route traversal, including arc driving and advanced multi-point turn capabilities in congested environments. The controlled stop planner provides the ability to stop the vehicle with a smooth deceleration. Finally, the multi-vehicle formation planner[5] allows for formations including line-haul, and lateral offsets between systems to facilitate route clearance.

Below the maneuver level sits the reactive level. Currently only one reactive planner exists in this system. The reactive planner is responsible for ensuring safe navigation around obstacles at all times, while also attempting to achieve the desired goal state commanded by the maneuver level. In turn, the reactive planner outputs actuator commands to the drive-by-wire (DBW) level.

At the DBW level, a mux arbitrates between the output of the reactive level and teleoperated control from the OCU. The throttle, brake and steering actuator commands are sent from the mux to a low level controller. This low level controller attempts to track desired velocity and steering angle as accurately as possible while minimizing overshoot, oscillations, and steady state error. The low level controller then sends the possibly corrected commands to the actuator drivers.

Alongside the mobility stack sits the payload stack within the system architecture. Similar to the mobility stack, the payload stack is broken into different functional levels. At the top of the payload stack is the vehicle-level payload arbiter, which interfaces with the section level planner. This provides access to any payloads attached to the system as well as arbitrating access to those payloads between the section level planner and the OCU. The arbiter takes commands from the autonomous system and executes those as long as a user is not currently controlling the desired payload. Beneath that is the maneuver level where nodes exist that enable additional functionality for simpler payloads, such as pointing to a geodetic location, or tracking an area as the vehicle moves. These behaviors can be added or removed as needed to normalize capabilities of various payloads. Finally at the lowest level there is the payload driver which translates from ROS into the payload's specific protocol.

This system architecture also defines a standard way of displaying pertinent messages to the operator from anywhere in the architecture. Any planner that needs feedback from the operator, or needs to inform the operator of important information can use the defined messages and the OCU will display the information or prompt to the operator. Prompts for feedback include a list of preset responses that the operator can select.

3. DYNAMIC DISCOVERY

One important characteristic of the system is the ability to discover and incorporate a dynamic assortment of vehicles, from those loaded with planners and sensor processors to manually driven vehicles without even a drive-by-wire kit. This is facilitated by the use of ROS Multimaster_fkie,[6] which consists of two nodes in each ROS Master: master_discovery and master_sync. For the multimaster to function, each component intending to interface with the system must maintain its own ROS Master. The master_discovery node looks periodically for any other active ROS masters running on the common network, as well as to advertise itself. The master_sync node then registers and updates any topics and services meant to be synced from the discovered masters to the shared namespace. Should any of those previously discovered masters no longer be responding, they will be de-registered.

After discovery, additional information is still required by many of the system nodes. Therefore, as a supplement to master_discovery, each vehicle must routinely publish a standard identification message, including the vehicle's name, classification, and IP address, to a Multimaster synchronized topic. The OCU subscribes to this topic and when it detects a new vehicle, it queries it to obtain a list of its capabilities, specifications, pose, and payloads. The publishing of this message by each vehicle also serves as a heartbeat; if a lapse of ten seconds occurs, all subscribers are to assume that vehicle has been lost to the system.

4. OPERATOR CONTROL UNIT

Taking advantage of the architecture and dynamic discovery is the OCU, which is split out into three essential components: the individual vehicle data models, a vehicle coordination handler, and the visible front-end. The design of the front-end facilitates at-a-glance assessment of the current system state, with command tools easily accessible. However, radio bandwidth constraints and limited visual real-estate mandate concessions on how much data the OCU can receive and display.

4.1 Components

Each vehicle advertised by the dynamic discovery is encapsulated in its own data model stored within a dynamic data structure imitating the array of physical vehicles. A message broker interfaces with the individual vehicles and the system architecture using independent communication links in order to create these data models and keep them current. The broker governs incoming subscription callbacks and outgoing command submissions via these links. Comprising these models is both a local mirror of the vehicle's unique attributes, routes, sensor data, and payload descriptions, as well as a list of the command functionality applicable to that specific vehicle.

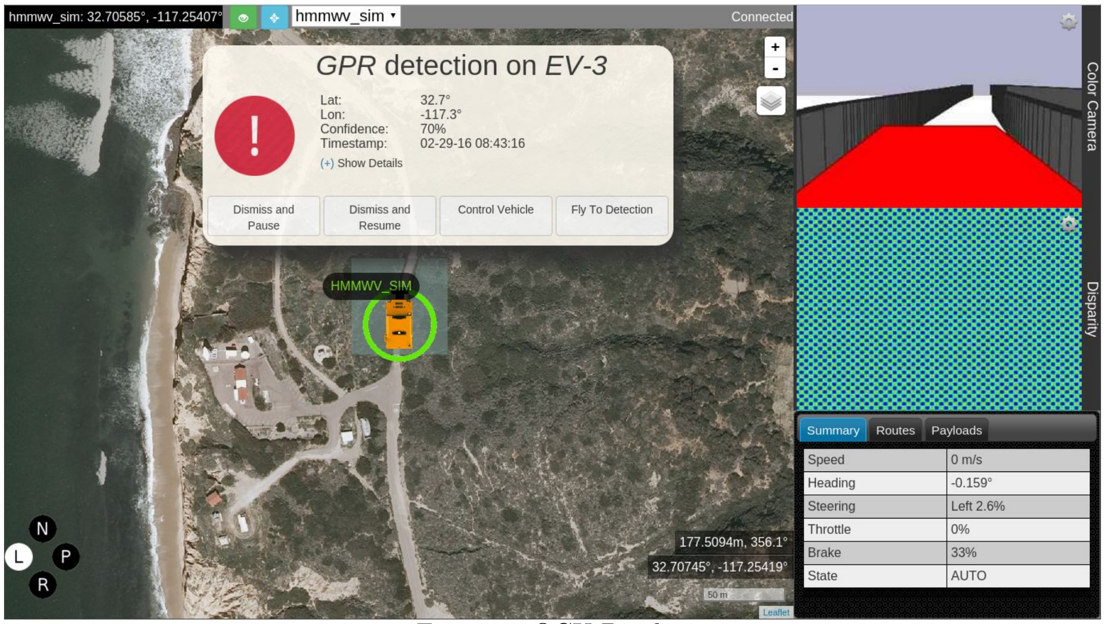
Beyond the broker and data models stand the tools needed to conduct multi-vehicle operations. Most high-level tasks need only be initiated and supervised by the OCU while the system architecture manages their execution. However, a handful of duties such as custom vehicle formations or coordinated tactical behaviors (CTBs) require a relatively large list of parameters that it is up to the OCU to deliver. Such efforts could also involve going through the broker for each vehicle involved in what would essentially be a series of low-level commands.

A collection of independent graphical user interface (GUI) modules make up the front-end. These reusable modules speed up both development and debugging by limiting the scope of the tasks each encompasses. Modules working with individual vehicles must get the data they need through the respective vehicle data model, and any interactions with that vehicle must go through the broker. Control mappings for joysticks are also modular and the active mapping can be swapped out at any time.

4.2 Design

The OCU is laid out in a quad display fashion as illustrated in Figure 2. Making up the quads are three GUI modules: a map, two vehicle video feeds, and a set of tools across a tabbed display. The layout of the quads can be dynamically customized to the user's changing priorities. The map is the primary means of conveying situational awareness to the user. The crucial components include aerial map tiles to give geolocation context, the pose, name, and autonomous state of each vehicle to discern where each member of the team is, and a stream of the currently controlled vehicle's costmap overlaid on the map tiles to know what areas that vehicle deems necessary to avoid. The map also serves as a canvas for working with routes and payload tools. The video modules allow the user access to any stream provided by the currently controlled vehicle and it's payloads. Anything else that is useful, but need not be continually before the user is relegated to the tabbed display. Tools provided are a vehicle status summary exposing data such as the currently controlled vehicle's speed and throttle, a menu for creating, modifying and submitting routes, and a payload manager.

Accompanying the quad display is a thin status bar at the top featuring a vehicle selector that lets the user quickly request control of any vehicle in the list, buttons to center the currently controlled vehicle in the map or to keep its heading pointing up, and the current communications status between the OCU and the system. Also claiming the status bar as their home are the unhandled system notification and payload detection indicators.



GPR detection on EV-3

Lat:	32.7°
Lon:	-117.3°
Confidence:	70%
Timestamp:	02-29-16 08:43:16

(+) Show Details

Speed	0 m/s
Heading	-0.159°
Steering	Left 2.6%
Throttle	0%
Brake	33%
State	AUTO

Figure 2: OCU Display

4.3 Considerations

The OCU must balance potentially overcrowding the display with too much information against leaving out or obscuring details that an operator would have otherwise used to make a decision. Preparation for the route clearance operations highlighted this conundrum with the notifications and detections. These are generated by nodes in the system and are published to the OCU for the operator to manage. Often of great consequence, it is reasonable to immediately prompt the users and expect them to make sorting it out their highest priority. In practice however, users did not take kindly to anything that might interrupt their current task and preferred simply a conspicuous reminder to deal with it at their earliest convenience. This was compounded by the fact that the OCU is at the mercy of sensors possibly generating numerous false detections, or system nodes supplying notifications of information the user may already be aware of. The embraced solution involves indicators appearing in the status bar that, when clicked, open a pop-up containing details of the notifications and a means of responding to or dismissing them. Once the user has dealt with all of them, the indicator disappears from the status bar. In addition, there are markers on the map showing where each of these notifications occurred. These markers serve as a short term history for the operator and clicking on them will display the detailed pop-up once again.

The larger the group of vehicles, the more arduous the task of managing them becomes. Supervising even the five vehicles in the exercise would occasionally inundate a single operator, and drove the demand for multiple OCU capability. Considering each OCU builds its own set of data models via their own broker, the workload required to keep OCUs in sync with one another is greatly reduced. One major outstanding detail pertains to how handling notifications and detections from one OCU affects other OCUs that will be trying to alert users of these same developments. Rather than having OCUs communicate with each other directly, the current system requires a notification to have a unique identifier and to be continuously published to all OCUs until it has been resolved. A resolution occurs when any OCU responds to that particular notification. Each OCU will monitor these unique notifications and should they discover a defined span of time has elapsed with no further publication of them, the OCU will assume that it has been handled and remove it from their user's display. An improved method to be implemented at a later date is to have a separate database to which the notifications and payload detections are stored, each with a "handled" attribute. Every OCU will monitor this database and in addition to easily being able to tell if a notification has been handled, they will also be able to display a richer history of all such notifications and detections on the map, rather than just the ones they have received since coming online.

While too much data may risk overwhelming the display, it may also risk saturating the radio bandwidth. It may seem excessive to aggressively unsubscribe from any topics that are not of immediate use, such as planner data for a vehicle that the user is not controlling, but across multiple vehicles over several OCUs, the combined bandwidth savings makes a significant impact. Furthermore, data that other nodes require at a high frequency (e.g. the DBW state), is throttled down when the OCU will see no benefit from that higher frequency. Another trick involving the costmap, which is essentially a monochromatic bitmap, is to encode it as a compressed mjpeg stream with a reasonable throttle.

Special consideration must be taken for video data, as it consumes the largest chunk of the available bandwidth. To begin with, each OCU is limited to simultaneously displaying a maximum of two streams plus the costmap stream. At this juncture, a balance of quality, latency, and bandwidth must be found for each stream. Based on feedback from users, the primary color cameras on the vehicle, which are heavily relied upon for teleoperation, demand the fewest concessions from quality and latency and therefore require higher bandwidth. The remaining feeds, including the disparity, infrared, and payload cameras, as well as the costmap, still seem able to adequately benefit the user even with lower quality, framerates, and higher latency. In the interest of simplicity and low latency, the current system relies solely on mjpeg video streams. The mjpeg implementation uses TCP which can result in a compounding bandwidth saturation in situations of poor comms should video or acknowledgement packets be dropped. However, there are plans to re-encode streams that are still useful even with higher latency using a more advanced codec such as H.264 or WebM over UDP, further reducing their required bandwidth at minimal cost to their quality.

4.4 MOCU

The OCU used for this exercise is built from the MOCU 4 (Multi-robot Operator Control Unit) project. MOCU is a multi-platform, web-based framework that provides many tools for controlling multiple robotic platforms. The modularity of the OCU is owed to the use of the AngularJS[7] and RequireJS[8] projects and keeping to their recommended style guidelines. These Angular directives enhance readability and are easy to leverage in other projects. An example used across nearly all versions of MOCU is the mapping directive built with the LeafletJS[9] and the ui-leaflet[10] libraries. For MOCU to communicate with the vehicles, two ROS packages are required: Roslibjs[11] and Rosbridge.[12] A Rosbridge node runs on each vehicle and provides a JSON (JavaScript Object Notation) API for non-ROS programs, such as MOCU, to have access to ROS functionality. The Roslibjs library is incorporated into MOCU and allows it to connect with Rosbridge over websockets.

5. CLOSING

As part of the exercise, operators were tasked with commanding behaviors and formations, monitoring the team of vehicles as they crossed over bridges, and handling payload detections. Concerning detections would require rerouting the system around the questionable region. In addition, the vehicles and payloads taking part in each of the four demonstration vignettes varied.

The OCU outlined in this paper was able to provide that required functionality. The system architecture supplied the means for the OCU to leverage the team of vehicles. Dynamic discovery handled the various vehicle and payload configurations throughout the exercise. Finally, the OCU provided the operators with situational awareness, and furnished them the tools to command the system.

REFERENCES

[1] Albus, J. et al., "4D/RCS: A Reference Model Architecture For Unmanned Vehicle Systems Version 2.0," in [*NISTIR 6940*], National Institute of Standards and Technology (2002). Gaithersburg, MD.

[2] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., "ROS: an Open-Source Robot Operating System," in [*ICRA Workshop on Open Source Software*], (2009).

[3] ROS.org, "pluginlib." http://wiki.ros.org/pluginlib/ (May 2015).

[4] Croix, J. P. D. L., Lim, G., Rahmani, A., Droge, G., Xydes, A., and Scrapper, C., "Mission modeling, planning, and execution module for teams of unmanned vehicles," Proc. SPIE 10195, Unmanned Systems Technology XIX (2017).

[5] Droge, G., Xydes, A., Ono, M., Rahmani, A., Toupet, A., and Scrapper, C., "Adaptive Formation Control for Route-following Ground Vehicles," Proc. SPIE 10195, Unmanned Systems Technology XIX (2017).

[6] S. Juan and F. Cotarelo, "Multi-master ROS systems," tech. rep., Institut de Robòtica i Informàtica Industrial (January 2015). http://www.iri.upc.edu/files/scidoc/1607-Multi-master-ROS-systems.pdf.

[7] AngularJS.org, "AngularJS." https://angularjs.org/ (2017).

[8] RequireJS.org, "RequireJS." http://requirejs.org/ (2017).

[9] LeafletJS.com, "LeafletJS." http://leafletjs.com/ (January 2017).

[10] github.io, "ui-leaflet." http://angular-ui.github.io/ui-leaflet/#!/ (2017).

[11] ROS.org, "RoslibJS." http://wiki.ros.org/roslibjs/ (April 2015).

[12] ROS.org, "Rosbridgesuite." http://wiki.ros.org/rosbridge_suite/ (June 2015).