Design and Implementation of API and GUI Interfaces for a Multiple Unmanned Autonomous Robotic System

Ahmed Barnawi, Fuad Bajaber, Abdullah Al-Barakati, Seyed Buhari, Anas Fatouh and Omar Alhubaiti

Faculty of Computing and IT, King Abdullah University, Jeddah- Saudi Arabia.

Abstract

Multiple Unmanned Autonomous Vehicle Experimental Testbed (MAUVET) is a state-of-the-art testbed that is being developed to facilitate research on engineering multiple agents robotic applications. Such testbed is essential to ensure quality of the application and satisfactory service delivery. In this paper, details of the testbed API and GUI interfaces are presented as designed and implemented. This system is made up of multiple autonomous and heterogeneous robotic agents performing a cooperative mission where each of them is assigned with a specific part in the mission proportional to its functions or capabilities. The system operator is involved throughout the mission i.e. to design the mission, to monitor the mission and to control the mission in some events evolving during the execution. Such provided privileges requires tight control over the system realized though the Graphical User Interface (GUI) that interact with the agents via a set of Application Programing Interface (API). In our testbed both GUI and API are developed from scratch with the main design objectives that the system’s software are based on modularity, reusability, interoperability and configurability. Important aspects of the system design were highlighted and performance evaluation of core components were also investigated.

Keywords: Unmanned Aerial Vehicles, UAV Controller, Embedded Systems, Multilayered architecture

I. INTRODUCTION

The Multiple Unmanned Autonomous Vehicle Experimental Testbed (MUAVET) is an experimental testbed of novel multi robotic system that supports communication and task distribution in real conditions of open-air area. The variation scenario targets a setup of multiple autonomous vehicles (AV) including unmanned aerial vehicles (UAV), ground vehicles (UGV) or marine vehicles (UMV), generally noted as UxV, performing coordinated tasks such as search scenario for objects of interest over a given area, Figure 1. The testbed contributes toward a wide range of benefits especially in case of application development. Such applications of multiple robotic system could include surveillance applications or any specific purpose automated applications based on mobile robots. Those applications require, beside onboard image and data processing, path planning and mobile robot’s traversal.

In fact, the literature doesn’t provide much detailed open design of similar platforms. It is usually the case that the underlined designs of these platforms are being tagged to robotic vendor SDK’s and licensed software libraries. In this paper, we summarize our findings on the design and implementation of the system’s API and GUI interfaces. In other related activities [1][2], we have introduced our contribution with respect to the traversal path planning for a search and find application.

Modularity is defined as the degree to which a system's components may be separated and/or combined. In software engineering, modularity implies that software is divided into different modules (components) based on functionality [3]. Each module has its inputs and outputs clearly defined, and software should be compartmentalized such that it allows for components to be developed independently with least amount of major design decisions impacting other modules in the system [4].

Reusability is another important design concept when it comes to complex software solutions development. In [5], it is defined as "...the use of existing assets in some form within the software product development process; these assets are products and by-products of the software development life cycle and include code, software components, test suites, designs and documentation.

The work in this project included various work packages such as the design and implementation of API and GUI. As design objectives, the intended interfaces have to ensure modular and reusable design of the software components and/or modules to support general-purpose use of the testbed and to minimize the customization effort needed to develop different application thus maximizing the ROI benefits of such a complex system. The structure of this paper is as follows; in Section II, related work motivation are introduced. In Section III, an overview on the MAUVET system is introduced. In Section IV, the system software architecture is presented. Section V, will discuss and classify the main API functions and socket interfaces. In Section VI, the GUI interface design and modules are detailed. In Section VII, we discuss the performance evaluation of the main components of the GUI’s Code Generation Engine (CGE). In section VIII, the conclusions and future work in this project are put down.

II. RELATED WORK

MASON [6] project involves the simulation and visualization library for JAVA that is able to handle numerous agents even in most demanding situations (such as swarm tactics). It has a modular design and modules can be added and or removed easily. The paper showcases separation of GUI and model and it mentions that this is what hampered some previous systems (TeamBots, SWARM, Ascape, and RePast). For that, this
software is platform independent. Design goals include speed, ease of modification, and platform independence. There are three layers in this architecture (utility, model and visualization). Utility layer concerns with general purpose, reusable components, the model layer models agents and events and their attributes in various classes. Last layer is the GUI.

On another hand, the work in [7] presents just the ground control station based on Robot Operating System (ROS). The design goals include using as much open components as possible, handling heterogeneous UxVs, developing a GUI monitoring and controlling component, and utilizing automatic flight path generation. This software can be run in Hardware in Loop (HIL) type simulation, or real experiments mode, which is our concern. This system uses different components from different parties and developed protocols for communication between every two components. GUI wise, design goals include ability to modify object properties mid-flight, real-time mission monitoring, and adaptability to varying number of UAVs. Multiple components are used here as well for path planning, map visualization etc. This software is an example of how using multiple off-the-shelf or configurable software would look like. Pros include short development time and a powerful visual capability with lots of features. Cons include having to develop custom protocols to facilitate interoperability between various programs used.

Another related conurbation is documented in [8], this paper presents details of architecture of software used, a modular system handling different UAVs, sensors, controllers and missions. This system also uses many off-the-shelf components (both hardware and software) to handle micro commands (flight related, GPS, low level sensors, … etc) but not in GUI. The system eliminates Ground-Control stations, so UAVs have on-board PCs with custom OS (QNX, blackberry- used for autonomous driving) handling planning and collaboration between UAVs. The architecture of the system is multilayered by complexity (e.g. low level on controller, Trajectory tracking, Trajectory design, Planning and task allocation). Also it worth mentioning that the system is task based with allocation and collision handling by QNX scheduling providing sub-optimality-greedy. GUI wise, the system proposes three different GUIs; Piccolo’s for mission monitoring, team Control and a Process Monitoring GUI. In fact this system demonstrates how software can be developed to allow single user to control many UAVs as opposed to most current solutions of high user/UxV ratio without the need for a robust or dynamic GUI.

To conclude on this point, our developed GUI interface features similarity to the work presented in [6] in the sense that the underlying system can run independent of GUI, though in that case, the GUI depends on underlying control system and is not modifiable. In [7] and [add reference third paper], however, they use off-the-shelf components for various parts in GUI and system while ours uses custom built GUI. We also identify similar GUI functional modules as in [8]. Finally, our GUI was built to interact to multiple drones in real time like [7] with one advantage to our system that it deals with heterogeneous robotic agents.

III. MAUVET SYSTEM OVERVIEW (BARANWI)

The MAUVET project is an experimental testbed of novel multi robotic system that supports communication and task distribution in real conditions of open-air area. The varication scenario targets a setup of multiple autonomous vehicles (AV) including unmanned aerial vehicles (UAV), ground vehicles (UGV) or marine vehicles, performing a coordinated tasks such as search scenario for objects of interest over a given area.

The designed system infrastructure should be applicable to all types of robots, but UAVs are considered as the primary actor in the platform. UAVs (or other robots) with unique functionalities may be incorporated in the system.

Figure 1 shows the basic system components and the interfaces among them. Over an open air area, the robots are deployed. The UAV agents are “physically” connected to the Base Station (BS) via the access point in a star topology, nonetheless, the UAV agents are logically interconnected to simulate UAV to UAV link. It is noticed that different UAV’s are introduced including ground and marine robotic vehicles. This system is operated and managed via Graphical User Interface (GUI).

The MAUVET Agent subsystem consists of onboard PC, Flight controller, GPS receiver, camera, Wi-Fi Communication Module and sensors. Functionalities of the UAV Control software is provided through a low level standard TCP/IP socket interface. A C++ application interface (API) is built above the socket interface in form of C++ functions encapsulated by MAUVET Interface class. The application code can exploit UAV control functionalities simply by calling function from this API. Both these interfaces are available on the server as well as on the UAV onboard computer, so it is up to user to decide strategy of controlling the agent i.e. centralized controlled via BS or autonomous control via onboard PC.

![Fig. 1. MAUVET](image)

IV. SOFTWARE ARCHITECTURE

Basic software architecture of the MAUVET system is shown consists of a central server providing central point of communication and connecting all connected entities, namely user applications (including user interface) and robots.

© Research India Publications. http://www.ripublication.com
The MUAVET system server is meant to act as a central node connecting user applications with the control software running onboard robots/UAVs. Basic block architecture of the server is depicted in Fig. 2, having TCP user interface on the left side and robot UDP interface on the other side. The connection between user applications and server is realized by TCP socket interface however, the connection between the server and several agents is realized by datagram-based UDP protocol.

The MAUEVET server software is written in C++. The server software is designed to have minimal dependencies on external libraries.

The communication protocol between server and robots is designed according to the following requirements:

1. Temporary loss of connection should not affect the system functionality
2. The latency in message delivery to the agent should be minimal,
3. In case of temporary loss of connection, the messages to the base station should not be delayed

In order to fulfill the above requirements, packets discard has been allowed on the lowest communication level between the agent and the base station, in case packet delivery fails. This makes the UDP an optimal protocol. An additional advantage of the UDP is datagram oriented communication, which is probably simplest form for message processing.

For secure message delivery, additional mechanism is needed to deliver mission critical messages those are the messages carrying the robot commands (sent to robots from the base station). This is achieved by sending an acknowledge message to each received command by robot. The secure delivery is managed by the server. If there is no acknowledge received for the specific command in the specified timeout, the command message is resent. However, in case of a longer loss of communication with the robot, this approach will lead to buffering of sequentially sent user commands thus violating the requirements for minimal latency. This problem is solved by defining the rule that newly sent command override previous command.

Implementation of the TCP interface is encapsulated by CMuavetTcpServer class with the public interface shown in the

![Fig. 2. MUAVET Server’s communication software components](image)

![Fig. 3. CMuavetTcpServer Class interface](image)

V. API FUNCTIONS AND IMPLEMENTATION

Application Programming Interface (API) allows sending commands to UAVs and retrieving data from UAVs. There is a significant difference between sending commands to a UAV and receiving data from a UAV. A UAV command is typically sent and acknowledged as per explained above however acknowledgment of received data is being omitted for design simplicity and bandwidth conservation.
As per the scope of our project, we have developed about 28 API functions written in JAVA programming language. Table 1 lists the main API functions against their arguments to command the flying mission of an UAV. However as for data retrieval such as GPS position, battery status, CPU utilization, sensor information and other type of information generated onboard.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm</td>
<td>(No parameters)</td>
</tr>
<tr>
<td>Disarm</td>
<td>(No parameters)</td>
</tr>
<tr>
<td>TakeOff</td>
<td>(No parameters)</td>
</tr>
<tr>
<td>LandOnPosition</td>
<td>2D position (lat, lng)</td>
</tr>
<tr>
<td>Land</td>
<td>(No parameters)</td>
</tr>
<tr>
<td>HoldPosition</td>
<td>(No parameters)</td>
</tr>
<tr>
<td>FlyTo</td>
<td>3D Position (lng, lat, alt)</td>
</tr>
</tbody>
</table>

There are two possibilities to retrieve data from the drone that are (a) periodic polling and (b) callback functions. Periodic polling allows to receive most recent data on request. As there is no data buffering on the server, the polling needs to be frequent enough in case any data should not be missed. On the other hand, when the period of the calling process is longer data generation frequency onboard, some data are missed. The callback mechanism calls a user defined function when new data arrives. This approach allows data delivery without any unnecessary delay. The vulnerability of this approach lies in possibility to delay or block data receiving, when data processing inside a callback function takes too much time.

To highlight some implementation details of the developed platform, let us go through the low level message structure as shown in Figure 5. The message frame is made of 4 fields, the START_BYTE field to indicate the start of the message followed by 2 bytes (LENGTH HI and LENGTH LO) to indicate the length of the message, which will tailed by the END_BYTE to indicate the end of the message.

Structure of the message data in the MUAVET system is unified among all system components and data links. This comprises mainly TCP connection between user applications and server and UDP based communication between server and UAVs. These messages are wrapped into datagrams specified for every type connection when transmitted. The message always consists of the header, fixed for all message data types, and the variable data payload, as depicted in Fig.5. The message structure is same for the commands sent by users to robots as well as for the data sent by robots. Table 2, explains each field in the message data briefly. Please note that many details are not being mentioned due to space limitation.
VI. GUI INTERFACE DESIGN AND MODULES

The developed GUI interface is a multiple components modular system the intention is that with minimal customization this GUI interface may fit desired application. The GUI module’s components breakdown is summarized in figure 12.

System modularity is ensured as the main four modules namely; Mission Design, Manual Control, Command Generator Engine (CGE), and Mission Monitoring, each are intended for doing a specific set of related functions. These functions (manifested over multiple reusable software components) are independent from each other. Each component has its input and output clearly defined. The modules are in turn interconnected in the sense that output from one module and is fed to another as it is needed for operator convenience.

It is important to stress here, as a design objective, our system is essentially a customized solution with configurability in mind. That is system modifications or update are enabled without prohibitive effort. For instance, if we wanted to utilize the testbed’s GUI to handle a certain application, the customization task may be limited to few software components associated with the desired functionalities while other components remain as is. The rest of this section summarize the GUI’s modules and their components as they are shown in Figure [the big figure].

A. Mission Design Module

In this module, initial parameters of search operation are set, the output of this mission is trajectory waypoints for each agent participating in the mission. The input of this module can be the searched area boundaries, number of drones and their basic capabilities such as battery level. Figure 6 shows data flow within the module and I/O with the trajectory algorithm server. The Parameter Manager Function is responsible for managing the gathering of inputs of different nature such as:

1) Search area boundaries based on user selects boundaries of the target area from the map interface. The interface uses google maps API and it is written in java-script with a back-end in java.

2) Drone parameters such as max-speed, battery charge and camera parameters.

3) Mission Parameters such as wind speed, targets number, … etc.

The trajectory manager function acts as a client to communicate with the Algorithm Server using JASON liberty that unifies objects communication among different programing language. This function also export the trajectories to other system components.

In mission design module, multiple components can be changed to suit required use without much effort. A different mapping solution can be used while needing a small change to java-script wrapper. This module utilizes JSON which is interoperable between many systems/languages, which means communication with external algorithm servers of any kind will be the same. Also, due to the nature of needed data from such servers (trajectory points with little other information), our system can be easily adapted for using any other algorithm with little to no modifications.

B. Mission Control Module

This module, with interface shown in Figure 13, consists of two main functions; (1) viewers to track drones’ location on maps and their speeds etc., and (2) a Manual/Event controller to manually control drones and offer event handling. The Viewers Function utilizes Java Swing’s threaded GUI application elements, mainly SwingWorker, which starts a new task and executes it intermittently along with the main GUI thread. Multiple workers are used to handle sending and receiving of commands, tracking drone stats, and handling console outputs from both BS and API client.

The Manual/Event Control Function has the following components:

- A log extractor, to read mission logs and interpret various parameters for display on GUI.
- An “Add to CMD queue” component, which sends manual user commands to a command (CMD) queue. Commands are enqueuer here and executed at the CGE Module.
- An Event Manager component that is responsible for event detection (from mission log and as designed in Mission Design module). An event may occur while the execution of the mission is taking place and further actions is required by the system.

<table>
<thead>
<tr>
<th>Table 2: message data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROBOT ID</strong></td>
</tr>
<tr>
<td><strong>CLIENT ID</strong></td>
</tr>
<tr>
<td><strong>MESSAGE TYPE</strong></td>
</tr>
<tr>
<td><strong>MESSAGE SUBTYPE</strong></td>
</tr>
<tr>
<td><strong>SEQUENCE NUMBER</strong></td>
</tr>
</tbody>
</table>

![Fig. 6. Data flow in Mission Design Module](Image)

© Research India Publications. http://www.ripublication.com
Through the interface shown in figure 13, users can send basic commands to UAVs like ‘Take-Off’, ‘Land’, ‘Go-To’ a specific point, or to follow a given traversal trajectory (series of points to travel through).

C. Command Generator Engine (CGE) Module

The CGE generates API codes for user’s commands to control the drone’s features. It also checks generated code for error prior to sending them out to minimize erroneous traffic between GUI and the base station or the drone. Figure 7, shown below illustrates CGE’s different functions.

![Fig. 7. Data flow in CGE Module.](image)

Execution in this module flows through three consecutive functions as shown in figure 7: First, Command Generator Function reads points generated by Mission Design module, and writes sets of drone commands. To handle the expected large volume of mission generated data, Quick input software component utilizes Java’s new I/O (nio) library to perform reading points and writing instructions. Performance of this function is discussed in following section.

Command Checker Function checks code generated by first function. It is also responsible for checking manual command generated Mission Control Module. Commands are checked for conformity to the state diagram shown below (Figure 9), that is, each instruction will transition the drone from a state to another, and possibly then another state. A drone should be in a specific state in order for a given instruction to be able to execute. For example, a drone cannot fly to a destination when it is not armed or hovering midair after having taken off.

After that comes Command Handler Function, which is responsible for sending / receiving commands/stats to/from drones and updating stats in mission logs for GUIs to view. Part of event handling is done here through events being logged in mission log, then Mission Control Module detects them and sends corresponding handling commands to command queue where this module sends them to drones. Mission Log component is a component where feedback coming from drones and base-station is written. Other parts of the GUI system like module 2 and 4 access this log to retrieve relevant information. The same command parser component from Command Checker Function is used in Command Handler.

How Command Checker Works?

Command Checking is performed based on the state diagram presented in figure 9 which summarizes all possible states a drone can take during the mission execution. Diagram shows that some states are terminal (gray background) and others are intermediate (blue background). Transition from a terminal state to another should pass through an intermediate state except between ‘disarm’ and ‘landed’ states. There are three support files needed for Command Checking operation. These components(files) act now as a cohesive unit, but they were separated to allow for future adaptation to other scenarios. If such parts were hard-coded, then adaptability will be hard and limited. Samples of the files are shown in Figure 8. (Actual files contain numbers instead of command names and states to ease processing). Files can be summarized as:

1) Instructions file: file contains API commands to drones to be checked (input).
2) Transition Rules File: This file contains the transition rules from a state to another. Each allowed transition is written in a single line, for example, ‘Armed’ -> ‘Landed’, is an allowed transition for a drone.
3) Command Rules file: This file maps a command to all states the drone moves through as it executes the command. A transition starts and ends with a terminal state as mentioned before. For example, the third line from this file (in figure 8, middle file) shows that ‘Take Off’ command takes a drone (from its current state, which should be the terminal state ‘Landed’ as inferred from transition rules file), to the intermediate ‘Taking Off’ and then to the terminal ‘Holding Position’.
The flow diagram shown in figure 8 explains the logic behind checking process. A drone’s current state is stored in a temporary location (say, T). Then the encountered command’s transitions (from ‘Command Rules’ file) gets added to T. After that, the resulting series of transitions in T is checked, one by one, against ‘Transition Rules’ file. If transition is found, then it is allowed. Finally, drone’s current state is updated to last state in T. For example, 3rd line of commands file in figure 8, (in T, and assuming drone’s current state was ‘Landed’) becomes (‘Landed’ → ‘Taking Off’ → ‘Holding Position’). If checker finds out this is correct -which it is- then drone’s current state becomes ‘Holding Position’. The requirement that a drone goes from a terminal state, through a transitional state(s) -if applicable- and end in a terminal state is ensured by how ‘Command Rules’ file is written. If checking was done for an automatically generated mission, then checker also checks that command sequence should make all drones end in ‘Disarmed’ state.

We should note here that the ‘Command Generator’ and ‘Command Checker’ functions, can be configured to generate commands and verify the generated commands for any other application using different types of agents, e.g. UAV or UGV, etc. For instance, ‘Command Checker’ can be used to check the validity of commands in different scenarios (state diagrams) just by modifying the support files as explained above. Note that to make checker work for some ‘goal based’ scenarios we need to modify the checking engine itself. An example of such a situation is when multiple paths are present out of a state but a drone needs to traverse a certain path to get to a ‘goal’ state and we consider other paths as wrong.

D. Mission Monitor Module

This module is assigned with displaying and presenting mission parameters using the information extracted from the data retrieved from the agent. This module’s functions utilizes the use of many functions from module 2 such as most viewers function components (text and map) while adding a video feed handler component to view footage and targets captured and detected by drones. Figure 14A, shows a screenshot of this module’s GUI design.

Data presented via the GUI interface in this module is retrieved from mission log while some others are calculated such as Percentage of mission progress; This is based on number of way-points traversed relative to all way-points in a mission. A more accurate way of calculating this would be based on distance traversed that is the total traversal distance is calculated by adding distances between every pair of waypoints and adding them to a cumulative-type array. This would give each traversed waypoint a different weight when calculating percentage. Time elapsed calculations is based upon the previous calculation by extrapolating time consumed for example if the first 50 meters or 20 way-points took 3 minutes to traverse, then the remaining 150 meters will take approximately 9 minutes to complete, this measurement will get more accurate as mission progresses.

VII. CGE PERFORMANCE EVALUATION (OMAR)

In the following section, we will address performance evaluation of the Command Generator Component as it plays a central role in the GUI design. The performance of this module is obvious to be a function of the processing power, in particular is the performance of the code generator. The performance of this component will be even more critical in case the commands controlling the drones are meant to be generated on board of a robotic agent (master) to command other agents (slaves). Our strategy for performance evaluation will be to calculate the processing overhead by running the code generator components on the base station server operated in two modes (full power and power saver modes).

For this testing run, and since, a random trajectory point generator was used to write trajectory coordinates that this component will use to generate commands. Three double numbers were randomly generated for each point (latitude, longitude, and altitude) and written in binary using Java NIO’s (New I/O) FileChannel which utilizes memory mapped buffers that mirror into a file for efficient, near hardware speed I/O. Times taken to generate the random sequence, read it back, and generate trajectory defining code were measured for a number of points ranging from 6,500 to 650,000 in increments of 1% for a total of 100 measuring points in both power modes (Full power @3.4 GHz and Power Saver @0.88GHz). Results show a 9-fold increase in time when number of points increases by a factor of 10 (at Full power) and ~3X increase when going from
Full power to power saver. The graph in Figure (10) shows a linear relationship between number of points and time needed to generate instructions to construct a trajectory. It is clear that relationship linear in both modes of operation.

Another aspect of the developed CGE to examine would be to check the validity of the Command Checker algorithm to make sure it produces correct outputs. To test this component, we generated many files containing sets of command sequences, some sequences are correct and some are not. By correct sequence, we mean that each command will takes a drone to a state that is reachable from drone’s current state as defined in the rules file based on the state diagram shown in Figure 9. An incorrect sequence of commands will inevitably violate this rule and the algorithm should catch this for further actions as explained above. Causes for the file being incorrect could be command deletion, insertion, replacement or any other malfunctions resulted from software instability. For all tested sets of command sequences, the algorithm was able to identify correct from incorrect. Figure 11 shows output from two testing runs, one involving a correct command sequence and other with an incorrect one, as in the 9th line when the ‘land’ command is replaced with ‘disarm’ command. At the 9th instruction, the state transition shows that the drone is in “HOLDING POSITION” state (i.e. state 5 in Figure 11) that implies that only three commands are expected to transit the drone to the next state according to the rules, those commands are FlyTrajectory, FlyTo, and Land however with the incorrect sequence, the 9th command was Disarm, therefore, violation is detected as it appears that the next state is undetermined and error message is generated “Sequence Wrong”.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a detailed design of our developed system’s API and GUI based on MAUVET platform. This platform contains BS connects multiple robotic agents to perform some user’s application. In this work, we have implemented more than two dozens of API’s commands to control the agent’s movement and retrieval of onboard data from the sensors or GPS receiver. We have shown that to ensure message success delivery from the robot to base station using UDP connection, a higher level acknowledgment mechanism is implemented. The GUI design on the other hand ensures system modularity to facilitate configurability and reduce the customization effort. We have presented our detailed design and dependencies between the GUI’s modules. We also detailed the design of some module’s functions and presented some performance evaluation results to appreciate some of the systems tradeoffs. The future work will include running experiments to further investigate the system’s performance under some extreme scenarios.

ACKNOWLEDGMENT

This project is being funded by the King Abdulaziz City for Science and Technology (KACST), under grant no. (AP – 35-162). The authors, therefore, acknowledge and thank KACST for its technical and financial support.

REFERENCES

Fig. 12: MUAVET GUI software components

Fig. 13: Mission Control Module GUI. (1) Map interactive viewer, (2) Console feedback from API, Base Station and drones, (3) Mission control, (4) Basic commands, (5) Trajectory management.

Fig. 14: Mission Monitor GUI. (1) Static map viewer, (2) Console feedback from API, Base Station and drones, (3) General mission stats, (4) Camera controls, (5) Mission stats for a specific drone.