Astrobee Robot Software: Enabling Mobile Autonomy on the ISS

Lorenzo Flückiger[1], Kathryn Browne[1], Brian Coltin[1], Jesse Fusco[2],
Theodore Morse[1], Andrew Symington[1]

[1] *SGT Inc., NASA Ames Research Center, Moffett Field, CA, 94035*
[2] *NASA, NASA Ames Research Center, Moffett Field, CA, 94035*
Emails: lorenzo.fluckiger@nasa.gov; kathryn.browne@nasa.gov; brian.j.coltin@nasa.gov;
theodore.f.morse@nasa.gov; andrew.c.symington@nasa.gov

ABSTRACT

Astrobee is a new class of free-flying robots designed to operate inside the International Space Station and perform surveying, monitoring, sensing and scientific tasks. Astrobee's capabilities include markerless vision-based localization, autonomous docking and charging, perching on handrails to conserve energy, and carrying modular payloads. Its open-source flight software runs on three interconnected smart phone class processors and uses the Robot Operating System. We present an architectural overview of the software and discusses lessons learned from its development. We highlight several projects already using Astrobee Robot Software to develop and test novel research ideas.

1 INTRODUCTION

Astrobee is a new class of free-flying robots (see Fig. 1) developed by NASA to perform a range of functions inside the International Space Station (ISS). Three Astrobees and a supporting dock will be launched to the ISS in late 2018.

Astrobee packs a tremendous range of sensing, computing and actuation capabilities into a one foot cube [1]. The open-source Astrobee Robot Software[1] drives these capabilities, running on three interconnected processors and interfacing with seven micro-controllers, six cameras, a propulsion system providing holonomic motion, and numerous other peripherals. This software localizes and navigates safely, autonomously docks and recharges, perches on handrails to conserve energy, accommodates external researchers' software, and more. The Robot Operating System (ROS) handles internal communication while the Data Distribution Service (DDS) communicates externally. The Astrobee Robot Software includes a Gazebo-based simulator and tools for building and deploying to embedded processors. In this paper, we describe the objectives and requirements driving the Astrobee Robot Software's design, the implementation approach, and the lessons learned from development.

[1] Available online: `https://github.com/nasa/astrobee`

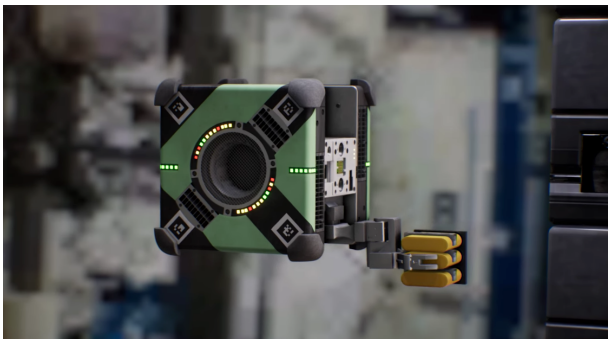

Figure 1. : A rendering of Astrobee in the ISS.

We take advantage of algorithms for robots on Earth, but also address challenges unique to the ISS:

- The **unconventional location** of and **limited space** within the ISS preclude localization techniques that exploit beacons, satellite navigation systems or Earth's gravity and magnetic fields.
- The ISS provides high bandwidth communications, but has frequent **signal loss and high latency**.
- The software must maximize **reliability** to minimize crew interventions.
- Astrobee robots are **not serviceable by an expert** on orbit, and thus the system needs to support completely remote maintenance and introspection.

2 RELATED WORK

Terrestrial aerial robots must overcome Earth's gravity and therefore have reasonably different constraints to free-flyers on the ISS. Nevertheless, they share similar challenges with Astrobee in terms of packing computing performance into lightweight form factors, localizing in unstructured environments and path planning.

Originally inspired by *AERCam* [2] and *PSA* [3], the *SPHERES* robots [4] are one of the most-used payloads on the ISS, and they have supported numerous micro-gravity research experiments over the last decade. The *SPHERES* robots therefore have limited computational resources. The *SmartSPHERES* [5] project extended *SPHERES* with a smart phone, demonstrating what sort of capabilities

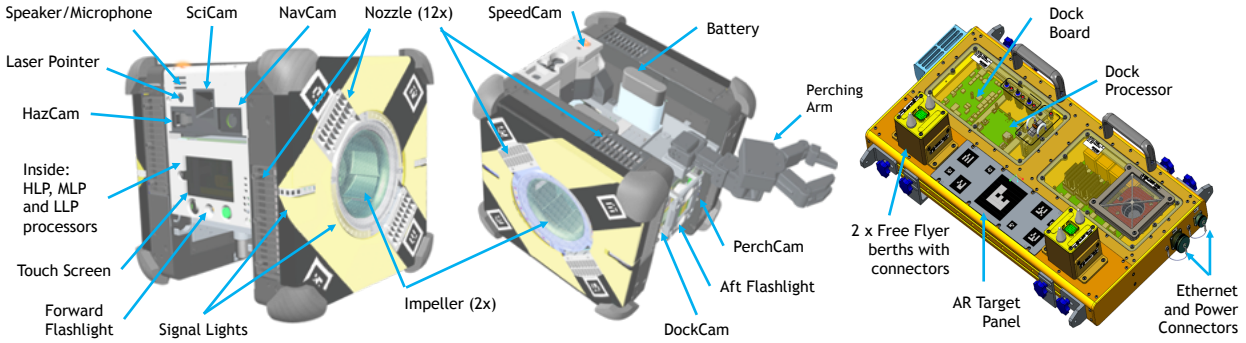

Figure 2. : Rendering of Astrobee equipped with a perching arm, highlighting the key external hardware components. On the right, the Astrobee dock (not to scale).

might be possible with a more powerful computing platform. Astrobee builds upon this work, providing a new class of free-flyer which provides substantially more computing resources, a greater level of autonomy and a new propulsion system that does not require CO_2 tanks or battery packs to be replenished.

More recently, the JAXA *Int-Ball* has successfully deployed a tele-operated free-flyer with remote camera capabilities [6]. *Int-Ball* is much smaller than Astrobee, and contains a miniaturized fan-based propulsion system that is commanded from a ground station. *CIMON* from DLR [7] is about Astrobee size, with a focus on delivering AI capabilities to assist Astronauts. Both *Int-Ball* and *CIMON* are currently designed to be permanently overseen by the ISS crew and are limited to operate in a single module.

It is worth noting that although NASA's *Robonaut* [8] is strictly not a free-flyer, the fact that it operates in the ISS means that is shares similar hardware, software and functional challenges with Astrobee. Robonaut uses ROS nodes deployed on multiple computers and relies on a vision system to perform tasks like grasping handrails.

3 THE ASTROBEE PLATFORM

The primary goal of Astrobee is to enable new research aboard the ISS: not only taking advantage of the unique environment without gravity, but also to study the role of robotics for space exploration. Astrobee is intended to support:

Research Platform: Enabling research partners to conduct scientific experiments in micro-gravity by developing their own software and/or payloads.

Autonomous Surveyor: Carrying payloads to perform spatial surveys of the environment. For example, an assessment of air quality or noise levels.

Mobile Camera: Permitting ground controllers to monitor crew operation with a high quality video stream.

The key software tasks required to support Astrobee's functions include:

- Localize throughout the U.S. Orbital Segment of the ISS without extra infrastructure.
- Precisely plan and execute motions without collision.
- Provide control and monitoring from the ground with resilience to communication loss.
- Support multiple control modes, including remote teleoperation, autonomous plan execution and onboard control by guest science (external researchers) software.
- Autonomously dock for battery recharging and wired communications.
- Autonomously perch on handrails to conserve energy while providing pan/tilt camera functionality.
- Manage guest science software, hardware payloads, and user interface components.
- Provide a control station application to command and monitor the Astrobee robot remotely (not addressed in this paper).

3.1 Hardware

Three Astrobees and one dock (see Fig. 2) will be delivered to the ISS. Astrobee may mate to one of two berths on the dock in order to obtain power and wired connectivity. Between the two berths is a fiducial marker comprised of a collection of augmented reality tags, which helps provide finer localization accuracy during docking. The dock is also equipped with a smartphone class ARM processor that monitors the robots' status when on dock and provides a mechanism for upgrading flight software.

Astrobee is equipped with three smartphone class ARM processors that communicate over an Ethernet network as shown on Fig. 5. The "Low Level Processor" (LLP) runs the pose estimator and control loop, and communicates with key hardware like the inertial, propulsion,

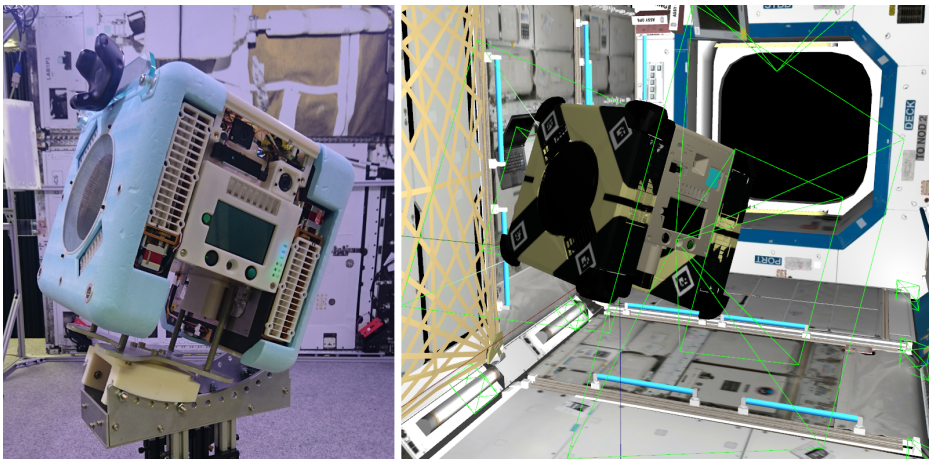

Figure 3. : Astrobee prototype on a granite table (left) for testing mobility in two dimensions, and simulated Astrobee model inside a virtual ISS (right) for testing mobility in three dimensions.

SpeedCam[2] and power management systems. The "Mid Level Processor" (MLP) is responsible for computer vision and mapping algorithms. Thus, the MLP is connected to two color cameras (*NavCam* and *DockCam*) and two depth cameras (*HazCam* and *PerchCam*). The MLP is also responsible for high-level control execution, managing faults and communication with the ground station. The "High Level Processor" (HLP) is dedicated to running *guest science* applications developed by research partners. The HLP also manages the human-robot interaction devices like the *SciCam*, touchscreen, speaker and microphone. The sensors and actuators locations are shown in Fig. 2.

Motion is achieved using 12 thrusters placed on two propulsion modules that sandwich the core computing module. Each propulsion module uses a battery-powered impeller to draw air in through a central intake, where it lightly pressurizes a plenum, providing air flow for six variable-flow-rate nozzles. The design using only two large impellers minimizes the sound level, but causes other control challenges like a latency of several seconds to achieve the desired fan rotation rate at startup.

Astrobees are nominally equipped with a perching arm that enables grasping of ISS handrails, transforming the robot into a remote pan-tilt camera.

4 SOFTWARE ARCHITECTURE

Astrobee relies on a modern software architecture: the system is composed of a set of modular, distributed, and loosely-coupled components. This is implemented in practice by ˜46 ROS nodelets[3]. The nodelets are grouped into ˜14 processes running across multiple CPUs. Dependencies between nodelets are kept to a minimum by strongly-defining their responsibilities and interface messages, services and actions. Fig. 6 shows the distribution of the main components on the Astrobee processors.

The LLP and MLP run Ubuntu 16.04 because of its widespread use and the availability of software packages, notably ROS. The HLP processor, however, runs Android (Nougat 7.1) because it is the only OS supporting some key hardware for Astrobee (the high resolution camera, video encoder and touchscreen). Android allows for the encapsulation of guest science software for the HLP as Android Packages (APKs), avoiding custom deployment and management methods. Astrobee does not require any real-time kernel extensions, since the control loop runs at a relatively low frequency of 62.5Hz. Actual servo and motor control is achieved with dedicated microcontrollers.

Software development is carried out in a virtual environment and code is first tested against a simulator (see right side of Fig. 3) before being cross-compiled and deployed to a prototype robot (see left side of Fig. 3) that operates on a granite table. The Astrobee Robot Software is mostly written in C++ because of its high-level constructs, ROS support, and runtime performance.

4.1 Communication Middleware

The various Astrobee software components rely solely on the ROS communication framework to exchange information. ROS messages are used for data distribution, ROS services for simple instantaneous requests (i.e., turning a light on/off), and ROS actions for complex longer operations (i.e., a motion that has an indefinite duration). The use of ROS nodelets and judicious grouping of large data producers and their consumers permits zero-copy message passing. For example, the camera drivers pass images to the vision algorithms without going through the network layer. The nodelet concept is so essential to the Astrobee that we developed our own specialization that

[2]The SpeedCam independently enforces that the velocity of the platform stays within safety limits.

[3]From the ROS manual: "nodelets are designed to provide a way to run multiple algorithms on a single machine, in a single process, without incurring copy costs when passing messages intraprocess".

augments nodelet functionality with unified message naming scheme, lifecycle control for the nodelet, a heartbeat mechanism, and fault management. These features, which are typically expected of reliable space software, are thus automatically available to all components.

Communication between one Astrobee and the outside world – either to a control station or another Astrobee – does not rely on ROS. Instead, RAPID [9] and the Data Distribution Service (DDS) are used. RAPID has a heritage of controlling robots on Earth from space [10] and robots in space from Earth [5], while DDS allows for a much finer control of the bandwidth usage and offers delay tolerance through Quality Of Service (QoS) extensions. These QoS capabilities combined with autonomy enables reliable tele-operation over degraded networks [11].

4.2 Command Interfaces

Astrobee offers the following command interfaces that can be mixed during a single session:

Tele-operation from a control station located at either ground control or on-board the ISS.

Autonomous execution of a planned action sequence, uploaded from either ground control or the ISS.

Guest scientist control of the robot's behavior by an Android application running on the HLP.

To support the multiple control interfaces we use a common command dictionary. This dictionary is processed by translators that create a set of DDS commands (using the RAPID framework) to support external operator control, and a Java application programmer interface (API) for guest scientists. Fig. 4 illustrates how commands and telemetry flow to and from the Astrobee Robot Software.

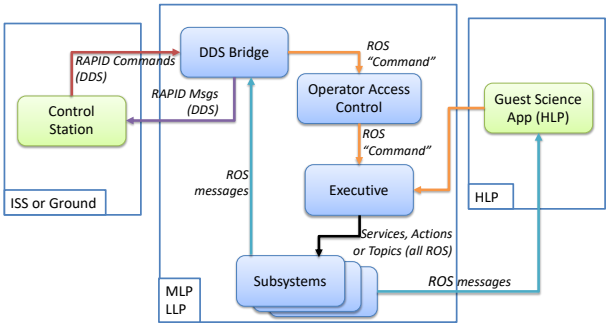

Figure 4. : Unified Command and Telemetry within Astrobee core system (LLP and MLP) and external applications (Control Station and/or HLP).

4.3 Remote Upgrade Mechanism

Astrobee software and firmware will periodically need to be updated on-orbit without astronaut support or external equipment. All the microcontrollers run a custom bootloader that allows firmware to be safely updated from the host processor. The HLP processor running Android is updated via network from the MLP using Android *fastboot*. The MLP and LLP processors have a more complex update mechanism using a rescue partition that is updated by a wired network connection to the dock. The Linux image creation for the ARM processors is based on a set of custom tools that guarantee consistency between the development environment and the images running on the embedded processors. For reliability purposes, the MLP and LLP use a read-only file-system with robot-specific customizations, like IP addresses and hardware serial numbers, applied using an overlay partition. The update mechanism requires a complex network, coupled with elaborate routing and firewalls shown in Fig. 5.

4.4 Simulator

An essential part of the Astrobee Robot Software is the simulator. The simulator enables Astrobee developers and guest scientists to test software before deploying it on a real robot. The simulator simulates the robot propulsion system, perching arm, color and depth cameras, inertial sensor, and environment. The simulator is written as a set of plug-ins for the Gazebo robot simulator. Each plug-in mimics the real hardware by offering the same ROS interfaces (topics, services and actions) as the real hardware driver. For example, ROS messages commanding the propulsion system are consumed by a custom Gazebo plug-in that converts the commands into forces and torques by a high fidelity model of the propulsion system. The resulting forces and torques then in turn used by Gazebo to compute the motion of the robot. An ISS CAD model provides a visual environment for the simulator.

The simulator allows the control system to run at its target rate and simulates realistic measurements for the localization algorithms. The simulator can also run faster than real-time (10 times speedup on a desktop computer with a modern processor and graphics card). Our architecture allows users to transparently run all components either in simulation or on the Astrobee, or as a mix of simulation with hardware processor(s) in the loop.

4.5 Software Components

Fig. 6 show the main software components distributed on the three processors.

4.5.1 Management

The **executive** filters incoming commands as a function of their source and the current operating mode (teleoperation, plan execution or guest science). DDS commands issued from the ground are transformed into ROS commands by the **DDS bridge** (see Fig. 4). The DDS bridge also subscribes to useful ROS messages for real time monitoring, and transmits them to the ground at a controllable rate as DDS messages.

The Astrobee Robot Software uses a distributed fault management framework. Each subsystem is therefore re-

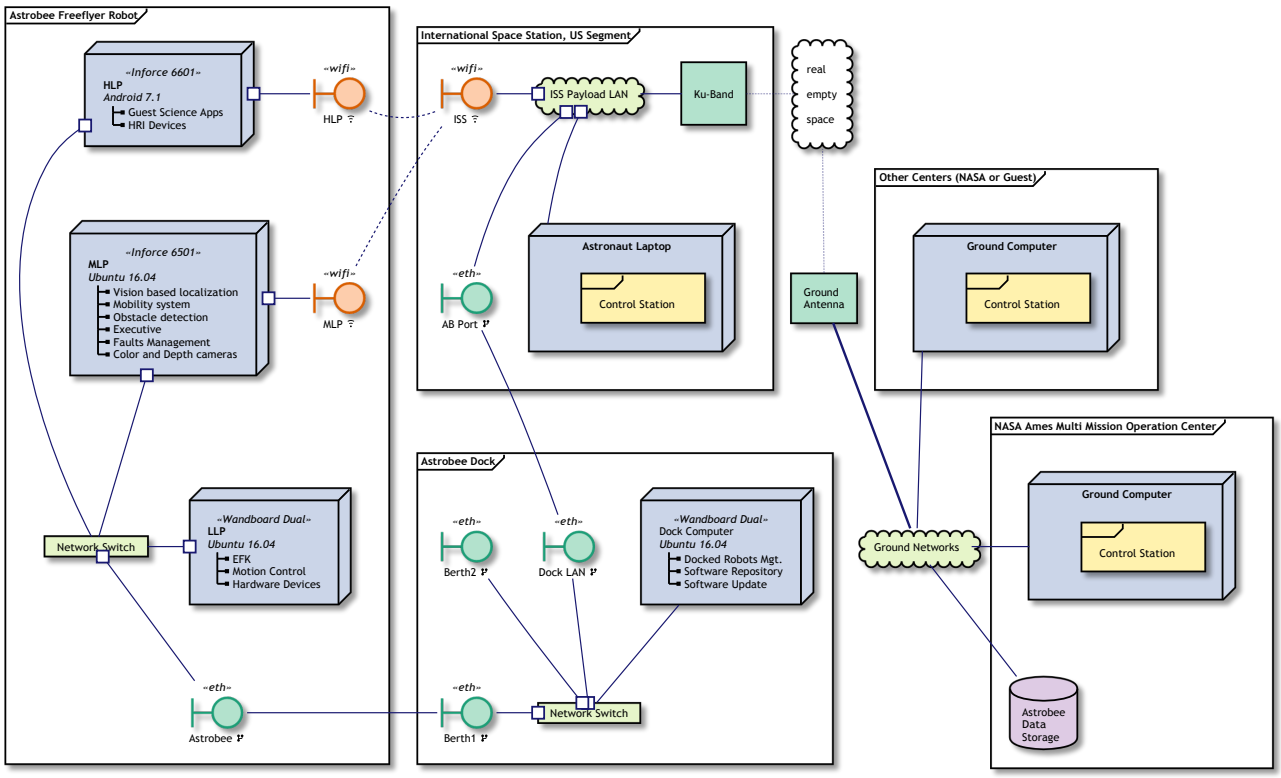

Figure 5. : Connectivity of Astrobee with the Dock and ISS network, including communication paths to the ground. Control stations can be used from NASA control centers, from partner institutions or from the ISS by an Astronaut.

sponsible for detecting and communicating its local faults to the system monitor. The **system monitor** uses information encoded in heartbeats from other nodes to aggregate fault information and responds according to a set of actions that are defined in the **fault table**.

The Astrobee Robot Software provides a common framework to support and manage Guest Science applications developed by external users. The **guest science manager** works in concert with the executive to control the life-cycle of guest science applications. The **guest science library** seamlessly integrates the guest science manager and the guest science APKs. The Astrobee Java API library encapsulates the command dictionary and allows guest science APKs running on Android to harmoniously communicate with the Astrobee flight software.

4.5.2 Localization

Astrobee's **localization** relies on a pose estimator that integrates measurements from a variety of sources depending on the robot's localization mode:

1. The general purpose localization uses the front-facing NavCam with a wide 120° field of view. Visual features from a pre-built map of the ISS results in a nominal position error lower than 5 cm.
2. When docking, Astrobee uses the aft-facing DockCam with a 90° field of view. The AR markers positioned on the dock allow a reduction in the localization error to a sub-centimeter level.
3. For perching on handrails, Astrobee uses the PerchCam depth camera. 3D features are extracted from the point cloud and fed to the pose estimator [12].

Additionally, a visual odometry algorithm tracks features across a history of images to improve localization stability. See [13] for further details about Astrobee's localization algorithm.

4.5.3 Control

Astrobee's motion control subsystem runs on the LLP. By limiting the number of processes running on the LLP, closed-loop control has a jitter well below the accepted tolerance. The control subsystem is developed using Simulink. C++ code is auto-generated with the Simulink buses being mapped to input/output data structures. The auto-generated code is wrapped into ROS nodes, so that it integrates seamlessly with the rest of the system. The parametrization of the control models uses the same configuration files that are used natively throughout the rest of the Astrobee Robot Software. This method allows Astrobee to easily translate the expertise of control domain experts into the ROS ecosystem.

The control subsystem includes three main components that are connected by ROS interfaces:

1. An Extended Kalman Filter (**EKF**) that implements the pose estimator described above.

2. A Control (**CTL**) loop that calculates a desired force/torque that drives the estimated state towards the goal state.
3. A Force Allocation Module (**FAM**) that translates forces and torques to propulsion nozzle commands.

This decomposition allows advanced users to replace a single component (typically CTL) with their own algorithms, while still benefiting from the other components.

4.5.4 Mobility

The **mobility** subsystem is responsible for planning, executing and monitoring all free-flyer motion. Trajectories can be synthesized on the ground using the control station, or calculated on-board using trajectory **planners**. The system uses a plug-in architecture to switch between path planners. The default trapezoidal path planner generates straight-line trajectories with trapezoidal angular and linear velocity ramps. One may also add an optional face-forward condition, which ensures the robot always faces forward as it moves. This condition ensures that the depth camera faces in the direction of motion, which enables the robot to "see" upcoming collisions over short horizons. A second Quartic Polynomial (QP) planner creates smooth, optimal trajectories around obstacles [14] and can be selected at runtime without affecting the rest of the system. The QP planner is appropriate for elaborate moves in free space, but not necessarily for simple linear moves that are required for docking or perching.

The mobility subsystem is also responsible for detecting obstacles. The **mapper** uses OctoMap [15] to build an octree-based occupancy map by fusing the HazCam depth camera with pose estimates. It then validates trajectories against this map and predefined keep-in and keep-out zones for each mission scenario.

5 LESSONS LEARNED

5.1 Balancing Research with Deliverables

Many components of the Astrobee Robot Software, such as the localization and planning algorithms, are novel in the field of robotics and are therefore the outcome of significant research. The innovative nature of the contribution means that work itself is unbounded – there are many research directions that might lead to an incremental performance improvement. Balancing the requirement to deliver a product with the desire to explore further research avenues is challenging.

5.2 Hardware Constraints

Astrobee, and in particular its propulsion system, is designed in-house across several groups at NASA using a combination of off-the-shelf and bespoke hardware. Developing hardware as a series of prototypes in lock-step with software necessitates communal access, which required careful planning and coordination.

Function	Coder	Eigen	Improvement
of_residual_and_h	87.0 s	2.0 s	98% (43x)
delta_state_and_cov	22.5 s	3.5 s	84% (6x)
covariance_multiply	18 s	< 2 s	89% (˜10x)

Table 1. : Performance of some time consuming functions using the Mathworks Simulink **Coder** compared to hand written code using the **Eigen** library. The times are totaled over the run of a recorded data set.

Furthermore, as a result of design constraints and hardware availability, the processing modules[4] within Astrobee differ from one another, with the exception of the LLP and dock processor. Even though various boards' kernel versions match, numerous customizations were required for each module. In addition, Astrobee runs both Linux and Android which forces the team to acquire expertise with two development environments and create different tools to maintain each system. This heterogeneity taxes software development effort.

Finally, the pace of the smartphone industry is much faster than that of a project like Astrobee, which spans multiple years. Products like processors and cameras become obsolete before the project is mature enough to commit to acquisition in sufficient quantities. This forces necessary software adaptations, which cost substantial time.

5.3 Software Optimization

The use of an embedded computing platform for computationally intensive algorithms necessitates optimization. For example, we replaced certain auto-generated Simulink C code with equivalent Eigen [16] implementations, which led to drastic performance increases which we show in Table 1. This is possible because Eigen exploits the accelerated NEON instruction set on ARM, whereas Simulink does not.

The MLP also offers a Graphical Processing Unit (GPU) that may be used to optimize other parts of flight software. Image processing algorithms are typically well-suited to this type of hardware acceleration. Unfortunately, drivers for this GPU are not currently available for Linux. However, in the future, we will consider using the GPU to improve performance.

5.4 Open Source Licensing

Astrobee is built using many open source libraries, and it has always been the project's intention to release Astrobee Robot Software under the Apache 2 license. The Intelligent Robotics Group's (IRG) push for open-source releases, coupled with NASA's contributions to the Open Source Robotics Foundation (OSRF) – the organization responsible for maintaining both ROS and Gazebo – demonstrates a commitment to engaging with a wider

[4] A processing module contains a CPU, GPU, memory and peripherals like I2C, USB and network adapters.

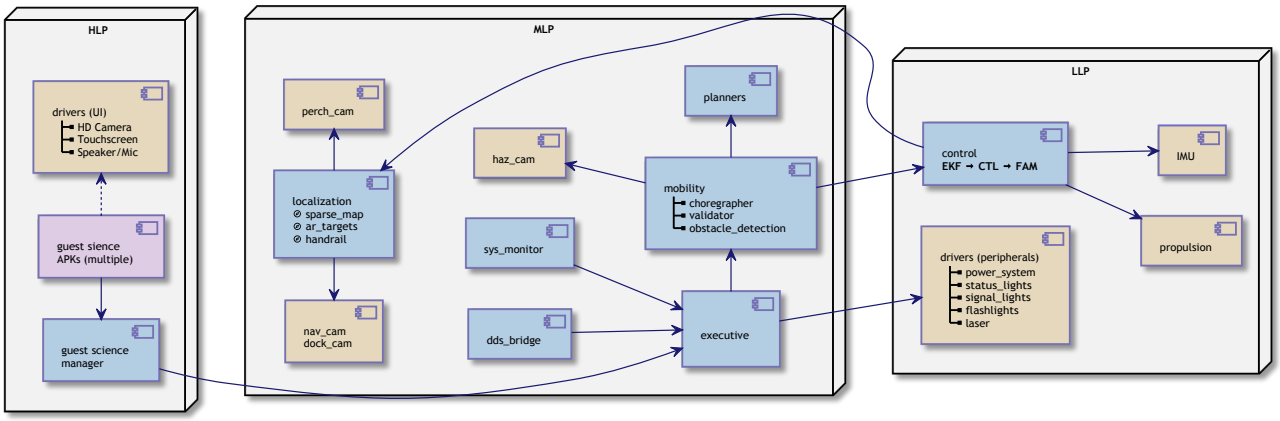

Figure 6. : The main software components running on Astrobee. The components on this diagram represent logical groupings normally composed of multiple ROS nodelets. The arrows indicate dependencies, not flow of information. Brown color represent drivers and purple external software.

community of developers.

The rigorous NASA open source process requires flight projects to be reviewed a comprehensive set of requirements [17]. The process however enables all existing or potential users of the Astrobee platform to view the code without signing licensing contracts with NASA.

5.5 External Users

The use cases below show how existing Astrobee Robot Software users are interfacing with the system.

5.5.1 Pure Simulation

The MIT Zero Robotics [18] competition enables thousands of students ages 12-18 from 18 countries to write code that controls robots on the ISS. ZR is transitioning from using the SPHERES free-flyer to the Astrobee free-flyer. In the first phase of the transition, ZR will develop a game framework using the Astrobee simulator and the Astrobee Java API without an Android system.

5.5.2 Control Algorithms

The Naval Postgraduate School in Monterey (NPS) is developing control algorithms for spacecrafts [19]. One goal of the NPS work is to enable zero gravity 'astrobatics', where a free-flyer with an arm can throw itself from handrail to handrail without using any propellant. In this scenario, the researchers replace the Simulink control with their own model. NPS also contributed an additional Gazebo plugin to handle the perching mode of Astrobee.

5.5.3 Hardware Payload

The REALM team at NASA JSC is developing a radio frequency identification (RFID) system for autonomous logistic management [20]. The REALM-2 RFID reader payload, leveraging Astrobee's motion capability, can pinpoint the location of lost items by monitoring RFID signal strength variations. REALM-2 will become Astrobee first hardware payload. For this scenario, the REALM team is developing a guest science application running on the HLP. To develop their system before the final Astrobee becomes available, the team acquired a HLP development kit to which they connect their hardware payload. The HLP development kit can be connected to a computer running the Astrobee simulator. This setup offers high fidelity testing with hardware in the loop.

6 CONCLUSION

The Astrobee Robot Software manages a powerful and complex hardware platform. Adopting both ROS and DDS as middleware enables us to build upon well-tested robotics tools while also respecting the unavoidable network constraints imposed by space. That being said, in the future we hope to transition to ROS-2, which by default uses DDS as a transport layer. Embracing ROS provides a reliable distributed system that integrates with reusable tools like Gazebo. We demonstrated that a properly crafted ROS based software system delivers a beneficial solution for an embedded robotic platform. The software update and network infrastructure for the Astrobee project is key for space deployment and maintenance and required substantial effort. The Astrobee robot prototype has been operated numerous hours without being hampered by the lack of real time operating system.

The software components developed enable:

- Markerless localization and navigation.
- Support for remote and on-board commanding, execution and monitoring of the robot.
- The ability to execute guest science applications that use a common interface to flight software.
- Flexible and efficient management of hardware resources with ROS enabled drivers.
- Faster than real time Gazebo simulation using plugins that mimic the real hardware.

The Astrobee Robot Software has been released as an open source project under an Apache 2 license. Not only does this enable guest scientists to develop experiments for Astrobee, but members of the public to obtain, test and potentially contribute back to the project. Three Astrobee robots will be commissioned onboard ISS in late 2018, and already more than 40 groups have expressed interest in using Astrobee. It is our hope that Astrobee Robot Software helps lower the barrier to experimenting with a new class of free-flying robot in microgravity, and enables exciting new research directions.

More information about the Astrobee project, including videos and the guest science guide, is available at: `https://www.nasa.gov/astrobee`

References

[1] Trey Smith et al. "Astrobee: A new platform for free-flying robotics on the international space station". In: *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS)*. 2016.

[2] Trevor Williams and Sergei Tanygin. "On-orbit engineering tests of the AERCam Sprint robotic camera vehicle". In: *Spaceflight mechanics 1998* (1998), pp. 1001–1020.

[3] Gregory A Dorais and Yuri Gawdiak. "The personal satellite assistant: an internal spacecraft autonomous mobile monitor". In: *Aerospace Conference, 2003. Proceedings. 2003 IEEE*. Vol. 1. IEEE. 2003, pp. 1–348.

[4] Swati Mohan et al. "SPHERES flight operations testing and execution". In: *Acta Astronautica* 65.7-8 (2009), pp. 1121–1132.

[5] Mark Micire et al. "Smart SPHERES: a Telerobotic Free-Flyer for Intravehicular Activities in Space". In: *AIAA SPACE 2013 Conference and Exposition*. 2013, p. 5338.

[6] JAXA. *First disclosure of images taken by the Kibo's internal drone "Int-Ball"*. URL: `http://iss.jaxa.jp/en/kiboexp/news/170714_int_ball_en.html` (visited on 07/14/2017).

[7] DLR. *CIMON - the intelligent astronaut assistant*. URL: `http://www.airbus.com/newsroom/press-releases/en/2018/02/hello--i-am-cimon-.html` (visited on 03/02/2018).

[8] Julia Badger et al. "ROS in Space: A Case Study on Robonaut 2". In: *Robot Operating System (ROS)*. Springer, 2016, pp. 343–373.

[9] Hans Utz et al. *The Robot Application Programming Interface Delegate Project*. URL: `http://robotapi.sourceforge.net/index.html` (visited on 05/01/2013).

[10] Maria Bualat et al. "Surface telerobotics: development and testing of a crew controlled planetary rover system". In: *AIAA Space 2013 Conference and Exposition*. 2013, p. 5475.

[11] Lorenzo Flückiger and Hans Utz. "Service Oriented Rbotic Architecture for space robotics: design, testing, and lessons learned". In: *Journal of Field Robotics* 31.1 (2014), pp. 176–191.

[12] Dong-Hyun Lee et al. "Handrail detection and pose estimation for a free-flying robot". In: *International Journal of Advanced Robotic Systems* 15.1 (2018), p. 1729881417753691.

[13] Brian Coltin et al. "Localization from visual landmarks on a free-flying robot". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE. 2016, pp. 4377–4382.

[14] Michael Watterson, Trey Smith, and Vijay Kumar. "Smooth trajectory generation on SE (3) for a free flying space robot". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE. 2016, pp. 5459–5466.

[15] Armin Hornung et al. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees". In: *Autonomous Robots* (2013). Software available at `http://octomap.github.com`. DOI: `10.1007/s10514-012-9321-0`. URL: `http://octomap.github.com`.

[16] *Eigen Overview*. URL: `http://eigen.tuxfamily.org/index.php?title=Main_Page#Overview` (visited on 03/20/2018).

[17] NASA. *NASA Software Engineering Requirements, NPR 7150.2B*. URL: `https://standards.nasa.gov/standard/nasadir/npr-71502` (visited on 11/19/2014).

[18] Sreeja Nag, Jacob G Katz, and Alvar Saenz-Otero. "Collaborative gaming and competition for CSSTEM education using SPHERES Zero Robotics". In: *Acta astronautica* 83 (2013), pp. 145–174.

[19] Josep Virgili-Llop et al. "Convex optimization for proximity maneuvering of a spacecraft with a robotic manipulator". In: *AAS/AIAA Astrodynamics Specialist Conference*. 2017.

[20] Patrick W Fink et al. "Autonomous Logistics Management Systems for Exploration Missions". In: *AIAA SPACE and Astronautics Forum and Exposition*. 2017, p. 5256.