JOURNAL OF AEROSPACE INFORMATION SYSTEMS
Vol. 18, No. 9, September 2021

Distributed Simulation of Heterogeneous Mission Subsystems Through the Black Lion Framework

Mar Cols Margenet,* Patrick Kenneally,* Hanspeter Schaub,† and Scott Piggott‡
University of Colorado Boulder, Boulder, Colorado 80309

https://doi.org/10.2514/1.I010827

This paper describes the design and implementation of Black Lion, a purely software-based, distributed environment for integrated testing of independent spacecraft mission models. The Black Lion simulation environment is architected to be reconfigurable, allowing for any number of heterogenous software models, across one or multiple computing platforms, to be integrated into a single spacecraft simulation. This architecture enables seamless integration of legacy software models that were never designed to work together into mission-wide simulations. A flat-sat scenario is used to showcase the capabilities of Black Lion. In this flat-sat scenario, Black Lion synchronizes and ties together the following components: a ground system model, a spacecraft physical simulation, and a flight processor emulator in which the flight software application executes. The numerical simulation presented in this paper showcases the closed-loop behavior of the entire spacecraft system.

I. Introduction

BLACK Lion (BL) is a communication architecture designed to enable distributed software simulation (SW-sim) of spacecraft (SC) systems. SC software undergoes rigorous levels of integration, validation, and testing. The coverage of these testing processes goes beyond the validation and verification (V&V) of nominal algorithm performance. Besides testing the nominal functionality and expected behaviors of an SC, simulations are critical to test off-nominal behaviors where components fail and sensor signals are corrupted. In addition, SC simulations are used to verify commands, communication protocols, and interaction with the ground. In overall, integrated SC simulations are key to test that SC flight rules are not violated, i.e., that prohibitive conditions or states on the SC are not induced upon execution of certain commands or onboard procedures.

The BL SW-sim is being developed in order to support flat-sat testing for an ongoing interplanetary mission in which the Laboratory for Atmospheric and Space Physics (LASP) and the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder are collaborating. Having said that, the BL architecture is designed to be a multimission and multiscenario software tool. In a flat-sat scenario, there are multiple mission components interacting with each other: the ground system (GS), the single board computer (SBC), and its flight software (FSW) algorithms, as well as SC models to simulate the dynamics, kinematics, and environment (DKE) of space. SW-sim testing uses virtual models or emulators in place of hardware components such as the GS or SBC. The idea behind an SW-sim is to provide a comprehensive simulation testbed that is purely software based. Figure 1 depicts the idea behind the virtualization of the GS, the SBC, and the SC's DKE. Ideally, the GS emulator should be able to ingest the same scripts as the real GS and should contain the same command and telemetry databases. On the same lines, the SBC emulator should run the unmodified FSW binary.

The challenge of an integrated SW-sim is that the different software components or models used are usually pre-existent resources from the particular mission that the SW-sim effort is supporting. These virtual components are stand-alone applications that are generally heterogeneous: they are written in different programming languages and they have different execution speeds (e.g., execution could be real time, faster than real time, or slower). Because of this heterogeneity, a dedicated communication architecture or layer is needed for the purpose of synchronizing the exchange of data between the different applications.

The BL communication architecture is designed to connect all the components or nodes of a distributed SW-sim while being as transparent as possible to the internals of these nodes, such that different mission users can plug and play their virtual models of choice. By design, the BL architecture obeys the Open-Closed Principle of objective-oriented programming, in the sense that new models can be added to the simulation without modifying the existing code. This principle is defined as follows: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." In this sense, while the BL development is currently motivated for an interplanetary mission, the system is being built under the principles of reusability and scalability. As such, the BL applications extend beyond the flat-sat example discussed here. Examples of further BL applications include the integration of large clusters of SC (i.e., formation flying), complex simulation components running on super computers or cloud servers, and, up to a certain extent, hybrid hardware-and-software simulations.

In the context of a flat-sat scenario, SW-sim testing does not replace hardware flat-sat tests. However, it can reduce bottlenecks by providing pure software substitutions for hardware components of limited quantity that might be needed simultaneously for testing by different mission groups. This alleviates schedule constraints and, in overall, it provides a flexible and cost-effective means of performing early-on and missionwide testing in a mission's program. SW-sim environments have long been used in the aerospace industry. Notable examples of aerospace missions using a software-only test-based approach are James Webb Space Telescope, Space Launch System, Juno, and OSIRIS-Rex. Different flavors of SW-sim architectures exist; some groups are developing their own in-house solutions like NASA's Operational Simulator (NOS) engine,[§] whereas other groups may have acquired and maintain versions of Lockheed Martin's SoftSim to support V&V activities. In the context of robotic Mars missions, there is also the Surface Simulation (SSim), which has been used on both Mars Science Laboratory and Mars 2020 missions to validate the sequences that will be uplinked to the SC [1]. In general, simulators tend to be sophisticated software products that are developed in parallel with the systems they are intended to test. However, all SW-sims are meant to provide a common functionality: the ability to run the unmodified FSW executable in a software-only simulation environment. A critical feature of the BL architecture,

Received 24 December 2019; revision received 21 December 2020; accepted for publication 25 May 2021; published online 14 July 2021.

*Graduate Student, Aerospace Engineering Sciences. Member AIAA.
†Professor, Glenn L. Murphy Chair, Aerospace Engineering Sciences. Associate Fellow AIAA.
‡Attitude Dynamics and Controls System Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics. Member AIAA.

§https://www.nasa.gov/centers/ivv/jstar/jstar_simulation.html.

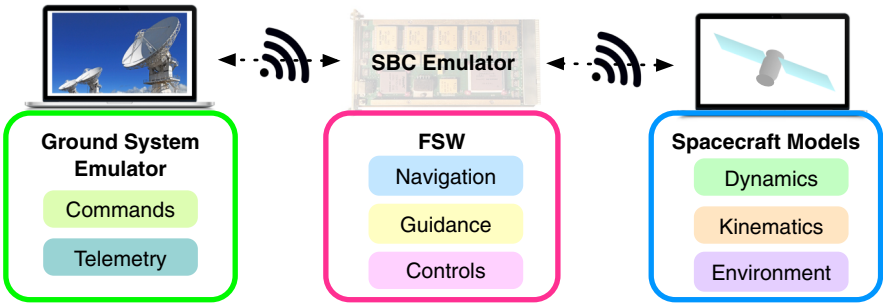


Fig. 1 Virtualization of spaceflight components.

which differentiates the system from the rest, is that it is meant to be not only mission agnostic but also fully open source.

The architectural characteristics of SW-sims strongly influence the functionality and, therefore, applicability of the simulation tool. With some generalization, simulation architectures are characterized by the degree to which system components are coupled. The coupling between simulation components is manifested by the simulation structure, where the overall system may be integrated as a single system of required components; integrated as a modular system with optional components; or developed as a group of cooperative yet stand-alone components.

An example of a system that has increased coupling between components is Advanced Solutions, Inc.'s (ASI's), Spacecraft Object Library in STK (SOLIS). SOLIS is a commercial plug-in to the Analytical Graphics, Inc. (AGI), Systems ToolKit (STK) mission analysis software. The plugin extends STK's orbit and space environment dynamics with the On-board Dynamic Simulation System (ODySSy). ODySSy is an on-board SC simulator providing additional models for rotational dynamics, sensors, actuators, power, and thermal dynamics, and basic SC control and guidance algorithms [2]. Using both SOLIS and ODySSy from ASI provides end-to-end SC simulation functionality. However, it does so by requiring those tools specifically. There are minimal options to substitute one component with another that was not intended to operate with the SOLIS system.

A software suite that demonstrates increasing modularity in its architecture is the NASA Jet Propulsion Laboratory's Dshell system [3]. The Dshell system avoids tightly coupled components by establishing interconnections and communicating data between components via connector signals. Connector signals allow each component to provide data to other components without requiring knowledge of the other components internals or availability [4]. The Dshell suite of components has grown since its initial development, and it currently supports a wide variety of simulation configurations, including both robotic and SC simulation, software and hardware-in-the-loop testing, and mission telemetry visualization [5].

The NASA Operational Simulator (NOS)[¶] is a simulation system that exemplifies the characteristics of a loosely coupled system architecture. The NOS system is a generic software-only simulation architecture and was developed by NASA's Independent Verification and Validation (IV&V) Independent Test Capability (ITC). The NOS system achieves its flexible architecture by employing a message passing middleware application to connect various simulation components by a virtualized MIL-STD-1553 or SpaceWire messaging bus [6]. This middleware approach allows users to add or remove heterogeneous simulation components, unique to a particular SC mission, without needing to rewrite or recompile model or application code [7].

Similarly to how NOS operates, BL integral simulations are developed as a group of cooperative yet stand-alone components that can run in a distributed fashion. The main difference is that BL is in the process of becoming open source (as intended since its inception), whereas NOS is an internal NASA IV&V tool.

The paper is outlined as follows: First, there is a preamble introducing the FSW application that is to be tested. Next, the different components to be included in the SW-sim are described. The following sections describe the communication process, the software tools adopted within BL, and the synchronization strategy. Next, a numerical simulation is shown, results are discussed, and several tasks are proposed as future work.

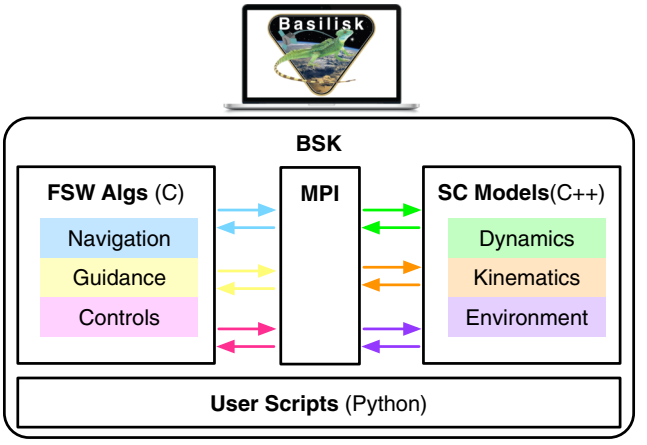


Fig. 2 Spacecraft models and FSW application developed within Basilisk (BSK).

II. Preamble: The FSW Application to be Tested

The collaboration between LASP and the AVS laboratory has produced a generic testbed framework for prototyping and testing flight algorithms. This astrodynamics framework is named "Basilisk" (BSK)[**] and is currently available as an open-source product [8]. BSK leverages Python's flexibility as a testbed for FSW development provided that the SC models and the flight algorithm code are written exclusively in C/C++, and then automatically wrapped into Python for simulation setup, analysis, and testing. The architecture of the BSK framework is depicted in Fig. 2. As illustrated, the BSK simulation system is decomposed into two main blocks: a high-fidelity simulation of the physical SC written in C++ (*SC Models* in Fig. 2) and GN&C algorithms written in C (*FSW Algs* in Fig. 2). All BSK modules are developed in a modular architecture using C, C++, and Python modules that communicate with each other through a custom message passing interface (*MPI* in Fig. 2).

The BSK desktop environment is used to construct and test different modes of the FSW application until the required functionality is achieved. At this point, the task, parameter, and state configurations are migrated out of the Python environment and embedded onto a flight target.

The flight target could be either a specific processor and Real-time operating system (RTOS) or a middleware layer like NASA's core Flight System (cFS). Targeting middleware is advantageous because it ensures portability of the FSW application among different processors and RTOS. Figure 3 depicts the case in which FSW algorithms are integrated within cFS and then embedded into an SBC

[¶]http://hanspeterschaub.info/bskMain.html.

[**]http://hanspeterschaub.info/bskMain.html.

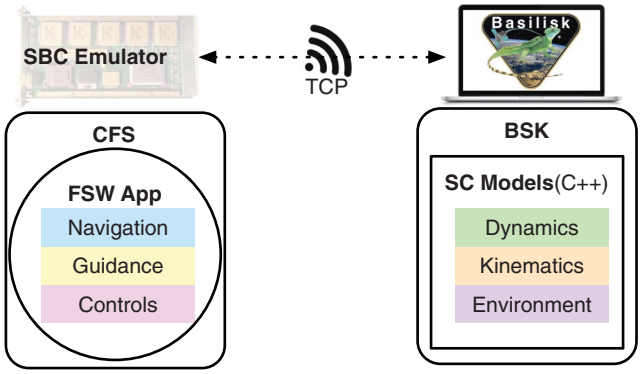


Fig. 3 Spacecraft models in BSK and FSW application migrated to a flight target.

emulator. A virtual SBC would be used in an SW-sim, whereas a physical SBC would be used in a hardware flat-sat system. Note that Fig. 3 also illustrates the distributed nature of the system, where communication between components happens through TCP connection. Previous work in Refs. [9,10] shows results of a peer-to-peer testbed systems in which only two components (i.e., a client and a server) could communicate. With BL SW-sim it is now possible to enable communication between an indefinite number of components.

III. Expansion of Simulation Models

Once the GN&C flight application is tested and stable, the next natural step is to expand the SW-sim coverage by including further components and analyzing the overall closed-loop behavior. Figure 4 illustrates the different components that are to be included in the BL SW-sim. As shown, the initial system with only the FSW application and the SC models is expanded to include also a GS virtual model, an SBC virtual model, and a visualization tool. Each of these models is briefly described next.

Ground system emulator: This includes the command and telemetry databases and runs the same mission scripts/sequences as the physical GS. It sends commands out and receives telemetry back, all in the form of Consultative Committee for Space Data System (CCSDS) packets. An example of GS emulator is the open-source COSMOS.[††] The BL SW-sim, however, uses a legacy model developed by LASP.

Virtualized single board computer: It contains the mission cFS-FSW executable, which runs on a standard RTOS. The SBC emulation also includes FPGA-like registers. These registers are emulated as a memory map for the input and output of raw binary data. An example of a processor board emulator is the open-source QEMU.[‡‡] The BL SW-sim currently makes use of a slightly modified version of QEMU.

Spacecraft models: The BSK simulation framework is used for both high-fidelity DKE models and hardware component models. The hardware models include sensors (gyro, star tracker, coarse sun sensors, etc.), actuators (reaction wheels, attitude control thrusters, orbit control thrusters), and the power control unit (PCU).

Visualization: A graphical user interface (GUI) is being developed with the Unity[§§] game engine in order to visualize the SC physical behavior as determined by the BSK SC models [11].

IV. Communication Between the Components

As mentioned earlier, the different components or nodes in an SW-sim tend to be stand-alone applications that are completely heterogeneous: they are written in different programming languages, wrap their internal data using different structures or packet types, and, in addition, present different execution speeds.

The differences between the particular nodes currently used in BL are illustrated in Fig. 5. The GS model is written in C++ and uses CCSDS packets with mission-specific data format. The SBC emulator is based on the open-source product named "QEMU." It is written in a combination of C/C++ and deals with raw binary data. The SC models are simulated within BSK. Although the BSK source code is written in C++, the application's interface with the external world is Python. The BSK packets are C++-defined structures. Finally, the Unity-based GUI is written in C#. The heterogeneity between all these components drives the need for a dedicated communication architecture, which is BL. The term "communication," as understood here, involves multiple goals:

1) *Transport* of binary data between nodes

2) *Serialization* of binary data (i.e., each node must know how to convert the received bytes into structures that can then manage internally)

3) *Synchronization* of nodes to keep all the nodes in lock step during a simulation run (lock-step synchronization implies that none of the applications runs ahead of the others in terms of simulation-elapsed time)

4) *Dynamicity* in the connections map (in the sense that none of the components should be required).

The purpose of BL is to achieve the described communication goals while being transparent and abstracted from to the internals of each node as much as possible. To achieve the desired level of abstraction the BL architecture has been designed as a single Central Controller and two APIs per node: the Delegate and the Router. The architecture is depicted in Fig. 6 and each of the components is described next.

Central Controller: This is the only static and required piece in the network (static in the sense that it has a fixed IP address). The Central Controller acts as a synchronization master and message broker between the components.

Delegate API: This interface manages sockets and connections with the Central Controller. It is the same script attached to all the nodes. The Delegate class is currently implemented for Python nodes, C++ nodes, and C# nodes.

Router API: This is a generic class with node-specific callbacks to create a custom interface to a legacy or newly developed mission simulation component. Its purpose is to route data in and out of the internals of the node. For instance, when routing out, the Router API gathers the node internal data, translates the data into a standardized BL format, and then passes the data to the node's Delegate API, who is ready to ship it across through the network.

Let us use a human language analogy to exemplify the functionality of the BL interfaces: Each node is an individual that speaks a different language (i.e., Spanish/French/German or, in BL, Python/C++/C#). The Router acts as a translator from the individual's language to a common standardized language (i.e., metaphorically English, and hexlified byte string in BL). If the Router is the translator, the Delegate can be seen as the communicator (i.e., the person who reads the translation out loud or, in BL, the interface that sends out the standardized data through the sockets). The final result is an English conversation in which each individual does not have to learn the particular languages of every other participant in the conversation. This property of the architecture is the key to its scalability.

V. Adoption of Modern Software Technology and Techniques

Before moving into the details of the communication hub, it is interesting to emphasize on the modern libraries that BL takes advantage of ZeroMQ (ZMW) and Google Protobuffers. These two libraries are briefly described next:

ZeroMQ Message Library: This is a high-performance asynchronous messaging library aimed at use in distributed or concurrent applications.[¶¶] It allows the transport of data to be fast, reliable, and protocol independent. The ZMQ interfaces are available in a wide

[††]http://cosmos-project.org.

[‡‡]http://qemu.org.

[§§]http://unity3d.com.

[¶¶]http://zeromq.org.

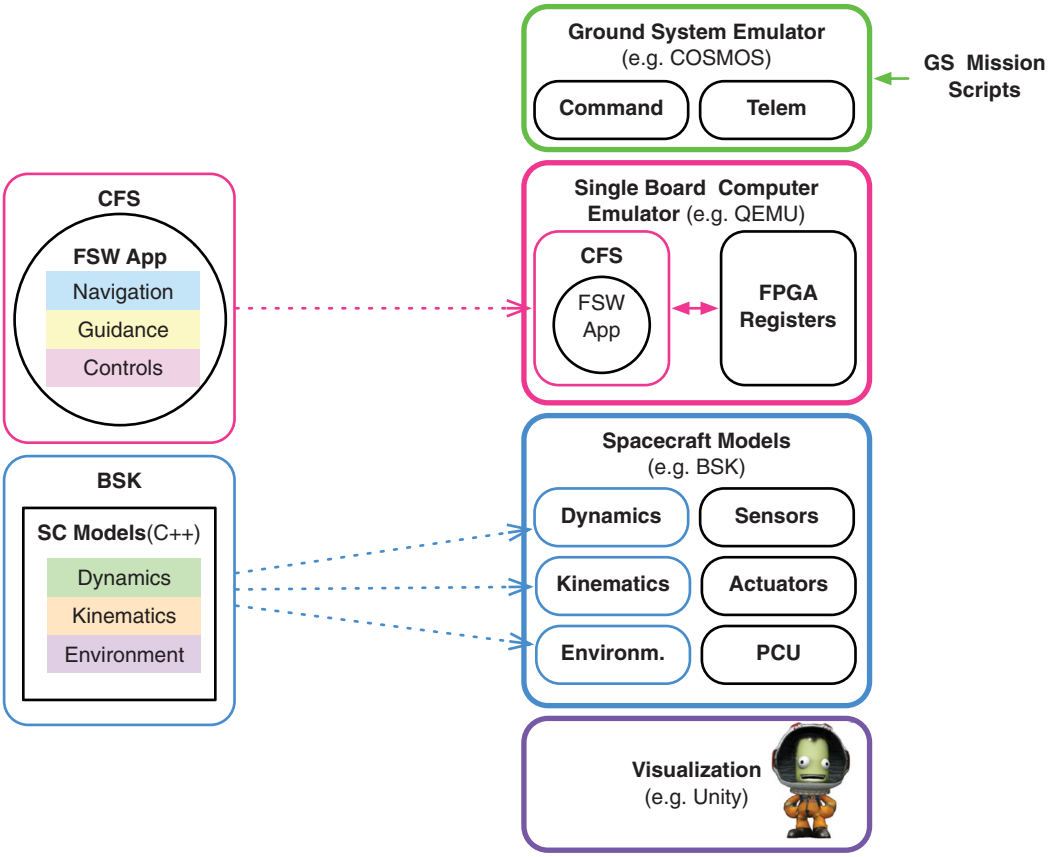


Fig. 4 Expansion from GN&C testing to an integral SW-Sim testing environment.

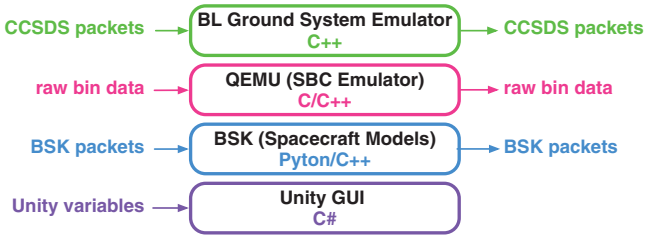


Fig. 5 Heterogeneity of programming languages and internal data packets in the BL SW-Sim.

range of programming languages, which can perfectly interact with each other.

Google Protobuffers: This is Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data (like XML, but smaller, faster, and simpler).[***] The user defines the structure of the data once and then it is possible to use special generated source code to easily write and read the structured data to and from a variety of data streams and using a variety of languages.

The adoption of ZMQ has been key to avoid bottlenecks in the BL broker architecture—in terms of both the number of nodes that BL admits and in the number of messages that can be sent across simultaneously. The reason for this is that ZMQ is meant for communication applications that operate in a much larger-scale and faster rate than what is demanded by an SC simulation.

While ZMQ is used for all data transport across BL, Google Protobuffers are only used for un/serialization in certain nodes. For instance, Protobuffers prove extremely useful for serializing and unserializing binary data that is shared between the BSK simulation and the visualization GUI. However, Protobuffers are not used in the

***http://developers.google.com/protocol-buffers.

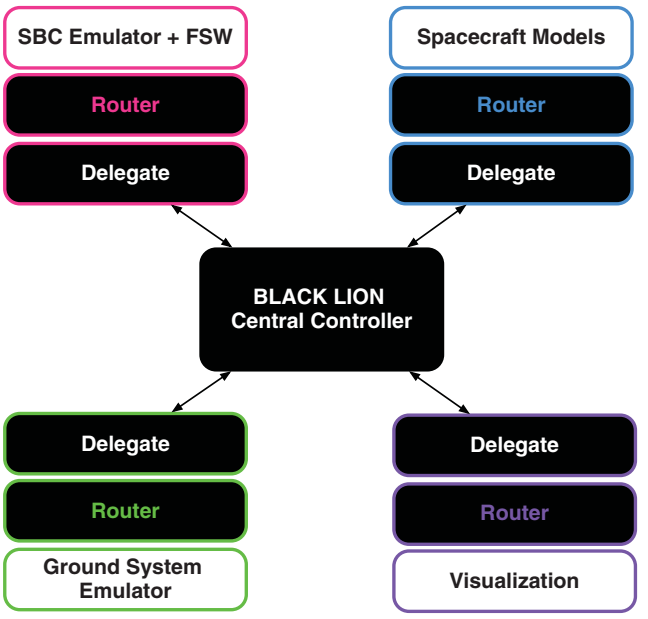


Fig. 6 Communication architecture: central controller, delegate APIs, and router APIs.

SBC emulator or the GS model because they would not reflect the operation of these components in actual flight.

VI. Data Transfer and Synchronization

A. Socket and Connection Definitions

To understand how the communication hub operates, it is necessary to explain first the socket patterns and connection types used in

the system. In particular, two types of ZMQ-socket patterns are used to transport data: the request-reply pattern and two different flavors of the publish–subscribe pattern. Their usage is described next.

Request (REQ)–Reply (REP): The central controller has an REQ socket for each node instantiated in the simulation that is used to make requests. In turn, each node has an REP socket that receives and parses the request, performs the commanded task, and replies back indicating accomplishment.

Publish (PUB)–Subscribe (FRONTEND SUB): Every node has a PUB socket to share its own internal data by publishing. In turn, the Central Controller has a frontend with a SUB socket that subscribes to the publications from all nodes.

Publish (BACKEND PUB)–Subscribe (SUB): Additionally, the Central Controller has a SUB frontend and a PUB backend. The messages received at the frontend are internally routed to the backend, which then republishes the data. In turn, each node has a SUB socket that subscribes to the messages of interest coming from the Controller's backend.

The relationship between sockets is exemplified in Fig. 7. The Central Controller is depicted in the middle, and two sample nodes are highlighted in magenta and blue (note that, within the nodes, the sockets are part of the Delegate API).

Now that the socket types are defined, the connections of these sockets to a given IP address and port are discussed. All the socket connections in the system fall into either one of these categories: static connections (i.e., binding type in ZMQ terms) and dynamic connections (i.e., connecting type in ZMQ terms). Only the Central Controller has a static connection, whereas each node's Delegate has a dynamic connection.

Central Controller: It is the only static piece in the network thanks to the frontend–backend (broker) approach. The controller acts as a server in the sense that it *binds* to a static IP address. Within the same address, it uses a total of $(2 + N)$ ports, where N is the number of nodes instantiated: one port for the frontend, one port for the backend, and a command port for each of the node-request sockets.

Nodes' Delegate API: Through the Delegate API attached to each one of the nodes, the nodes become dynamic clients that can come and leave without bringing down the rest of the system. This dynamicity is reflected in the fact that the nodes only *connect* to an address and port, rather than bind.

Through the described strategy, the Central Controller acts as a sever and it is always required, whereas the nodes are independent entities or clients that do not intrinsically rely on each other. The use of ZMQ also allows all the connections to be protocol independent (TCP, IPC, etc.)

B. Request–Reply Communication Between the Controller and the Nodes' Delegate

The requests from the Central Controller to each node's Delegate are not SC commands; they are communication and synchronization commands exclusive to the SW-sim. In the current BL implementation, the Controller can make two different types of requests: one-time requests or cyclic requests. Examples of one-time requests are the "Initialize" request (upon which a node performs its internal initialization and establishes sockets connections) or the "Provide Desired Message Names" request (which instructs each node to report all the messages that would like to receive). An example of cyclic request is the "Tick" request, which is used at every time step of the SW-sim run for synchronization purposes. This request contains the time duration of the next time step (i.e., Δt, which is currently a constant value although it could be variable). Upon receiving a "Tick" request, each node needs to perform a series of actions and then reply with a "Tock." More details on the "Tick-Tock" synchronization are explained in the next section.

C. Tick-Tock Synchronization

Three actions are performed by each node upon receiving a "Tick" request: *publish, subscribe*, and *step simulation* forward.

Publish: The node's Router collects the application internal data and makes them available to the node's Delegate for publication to the controller's frontend.

Subscribe: The node's Delegate receives external data coming from the Controller's backend and passes the data to the node's Router, who is responsible for writing these messages in the internals of the application.

Step simulation: For synchronous nodes (i.e., those who run in cycles like FSW rather than being event driven like the GS), "step simulation" implies executing for the commanded Δt in order to generate new data.

Because each node is an independent application with different execution speed, the "Tick-Tock" synchronization ensures that all of them are kept in lock step (i.e., no application runs ahead of another in terms of elapsed-simulation time). For instance, if one of the synchronous nodes finishes executing for a Δt earlier than another, it sends the "Tock" reply and waits for a new request from the Controller. Because the Controller will not proceed until it has received all the "Tock" signals from all the nodes, the SW-sim speed is naturally driven by the slowest component of the simulation. For asynchronous nodes (i.e., those that are event driven), stepping forward over a Δt is not applicable. Yet, between a "Tick" and a "Tock," they still publish and subscribe.

It is important to mention that, with the described "Tick-Tock" mechanism, the BL architecture only allows satisfying a single hard-time constraint, which also has to be the slowest one. When there is a hardware component in the loop that runs in real time (which would be a hard-time constraint), the remaining models must run real time or faster. For the emulated flat-sat scenario presented in this paper, only the processor board emulator imposes a hard-time constraint. The dynamics simulation can run faster than real time, whereas the GS and visualization are asynchronous.

VII. Numerical Simulation

This section showcases a BL numerical simulation in which three different nodes run in a distributed fashion. These nodes are the BSK SC physical models, the SBC emulator running the mission FSW executable, and the GS model. In this configuration, the Central Controller is responsible for synchronizing the aforementioned

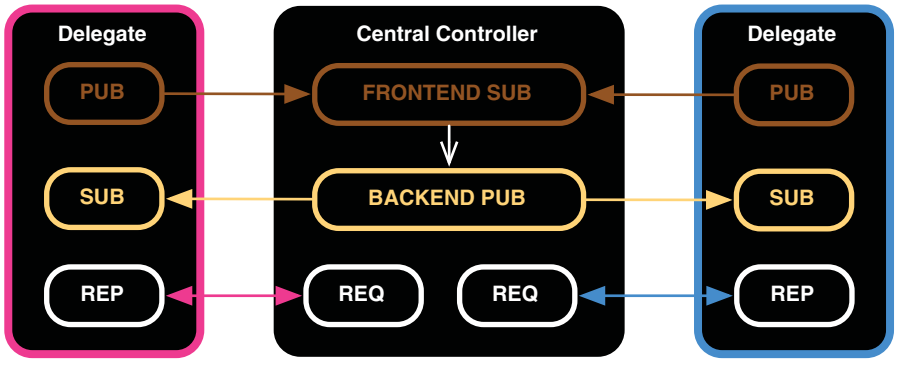


Fig. 7 Socket patterns between the central controller and sample nodes.

nodes and orchestrating the data exchange. This distributed simulation setup is depicted in Fig. 8.

A. Node Configurations

The integrated numerical simulation to be shown encompasses commanding the SC into different pointing modes. Because each node is an independent, stand-alone application, it is critical that all of them are configured for the same scenario. For the pointing guidance scenario considered here, each node is configured as follows:

Basilisk node configuration: The physical simulation is configured in a state where the SC probe has just separated from the launcher after leaving Earth's sphere of influence. The simulated true attitude and rate of the SC are set to the following values:

$$\boldsymbol{\sigma}_{B/N}(t=0) = [0.4, 0.2, 0.1] \tag{1a}$$

$$\boldsymbol{\omega}_{B/N}(t=0) = [0.0, 0.0, 0.0] \tag{1b}$$

where $\boldsymbol{\sigma}_{B/N}$ is a modified-Rodrigues-parameter (MRP)-based attitude description [12–14] and $\boldsymbol{\omega}_{B/N}$ is the inertial angular velocity. Hardware components modeled in Basilisk that are relevant in the present simulation include a set of four reaction wheels (which are used to control the SC) and a dual-headed star-tracker (which provides attitude and angular rate measurements).

Ground system configuration: The GS model contains the complete suite of command and telemetry databases for the mission. The graphical, interactive interface of the GS model can be used to send commands through BL and also to monitor telemetry. The commands could be either uploaded as block scripts or issued manually by the user. In the numerical simulation shown in this paper, the user manually sends a series of commands.

FSW configuration: The FSW executable runs as a cFS application on an RTOS in the SBC emulator. Four different registers have been implemented: two input/output board registers (IOB), the solid-state recorder register (SSR), and the SBC register. Through the FPGA registers, FSW reads and writes in a hardware-like fashion that also replicates interrupts. The idea is that each register has an associated memory buffer, and specific FSW states are mapped to specific addresses within these buffers. Not all the internal FSW states are mapped to the register's buffers, but only those that require interaction with the external world. The specific FSW states that are shared with the external world depend on the scenario being considered. Hence, increasingly complex scenarios demand the implementation of additional register spaces and their associated handlers. For a complete description of the implemented register spaces, the readier is pointed to Ref. [15].

B. Data Exchange and Synchronization

In the numerical simulation presented, 10 different messages are being exchanged across BL. These messages are illustrated in Fig. 9 and they are related to commanding the pyramid of reaction wheels (RW), processing attitude sensor data, parsing commands, and sending telemetry. The BL communication time step is set to 0.01 s, with all the synchronous applications being updated internally faster than this communication rate. The communication time step is reconfigurable and, in general, is set according to the FSW communication requirements for the scenario being tested.

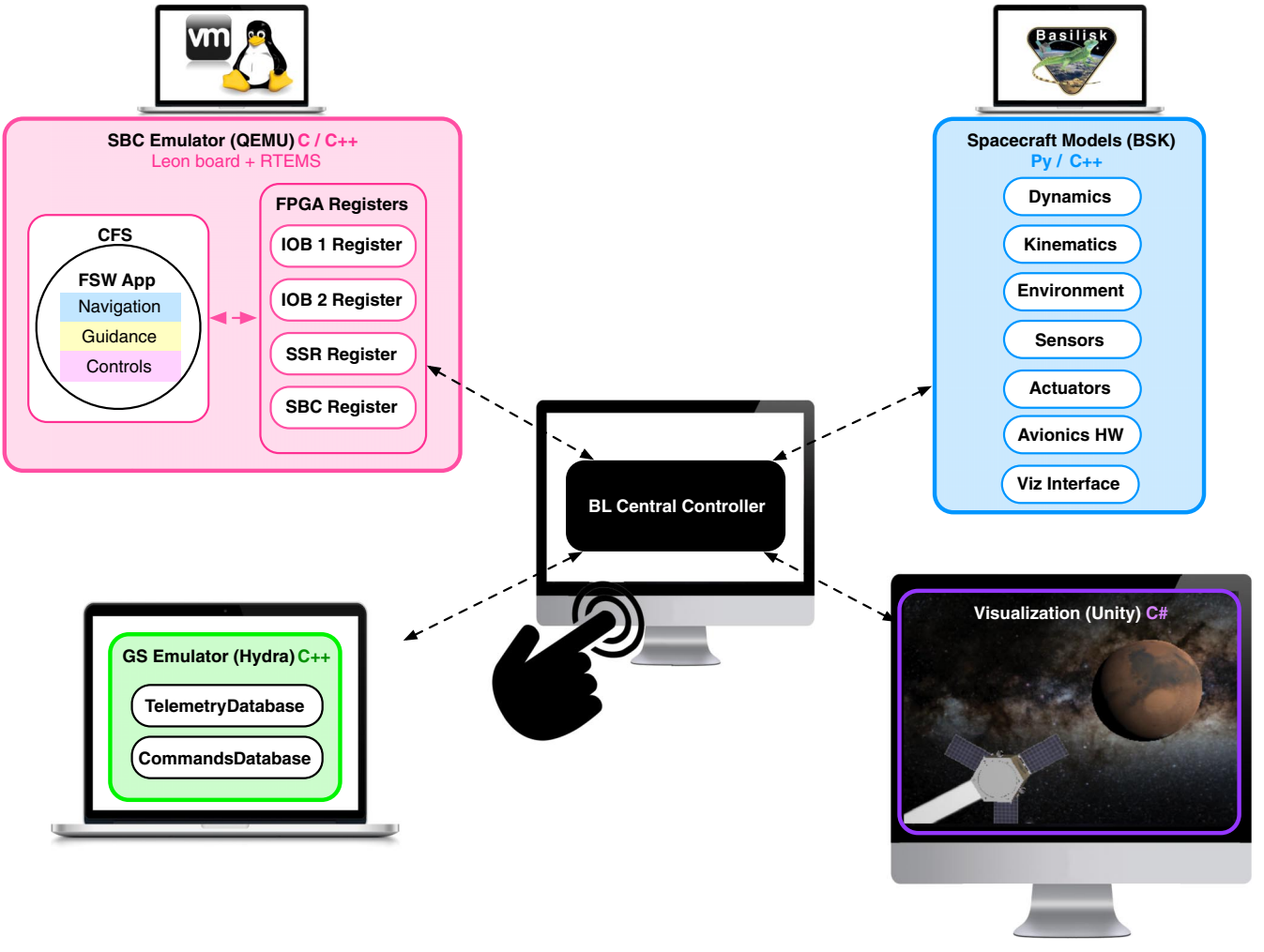


Fig. 8 Nodes in the integral SW-Sim run.

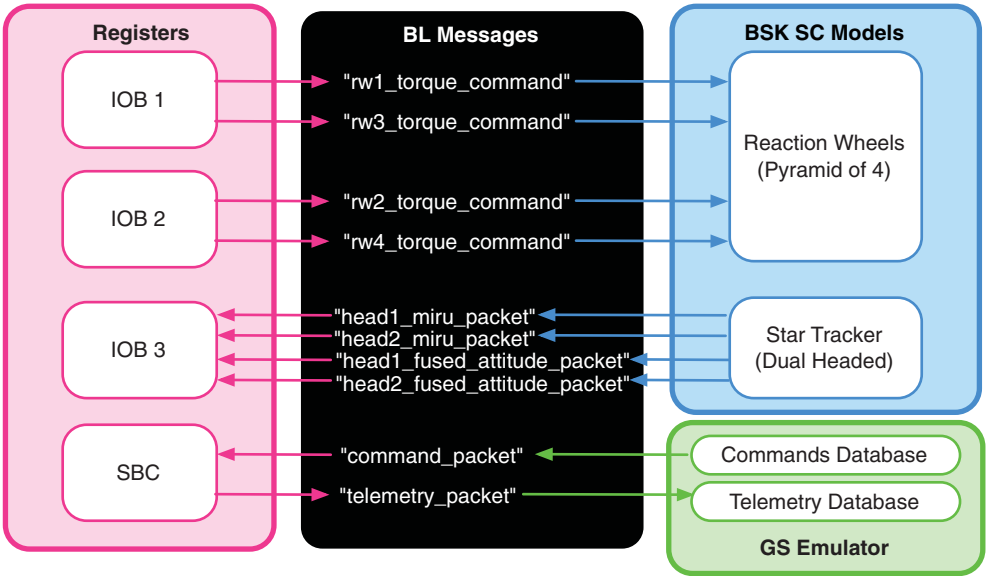


Fig. 9 Messages shipped through the black lion central controller.

C. Closed-Loop Numerical Results: Spacecraft Pointing Commands

As mentioned earlier, this numerical simulation consists on commanding the SC into a series of pointing maneuvers. In this case, the user uses the GS interface to manually send commands to FSW as well as to monitor the reported telemetry. The different commands issued by the user are the following:

1) Nav monitoring and inertial pointing command
2) Ephemeris correlation and Mars pointing command
3) Sun pointing command
4) Mars pointing command

These commands are stored in the FPGA registers and picked up by the cFS-FSW application in order to reconfigure the onboard pointing mode. In the meanwhile, sensor data from the SC physical simulation are being continuously updated within the registers. Once the nav monitoring command is received, FSW starts using the sensor data as an input to the navigation filters in order to achieve attitude lock. The user is able to monitor the attitude dynamics and controls (ADC) packets in the telemetry stream through the GS interface, hence keeping track of the current FSW attitude states. Once attitude lock is achieved and a pointing command (e.g., inertial pointing) is issued, FSW estimates the SC's current pointing attitude, derives the associated tracking errors, and computes the control torques required to drive the SC into the desired attitude. The control torques are stored in the registers and sent across BL to the SC physical simulation, where a set of four reaction wheels is used to apply the commanded control torque.

Figures 10–12 show the closed-loop response of the SC physical simulation. In particular, the plots in Fig. 10 correspond to the MRP attitude of the SC's main body frame; the plots in Fig. 11 show the reaction wheel speeds driven by the control voltage commanded by FSW; and Fig. 12 displays the miss angle (in degrees) of the onboard Mars instrument and solar arrays. These plots help testing not only the flight algorithm performance but also the validity of the GS

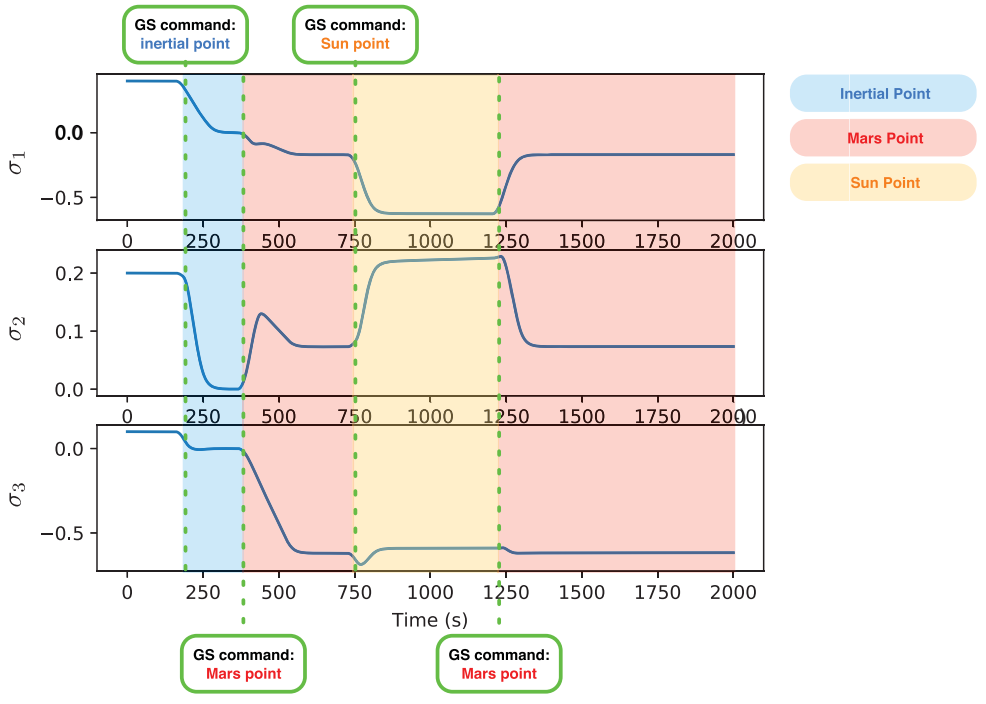


Fig. 10 Closed-loop response: spacecraft's main body attitude.

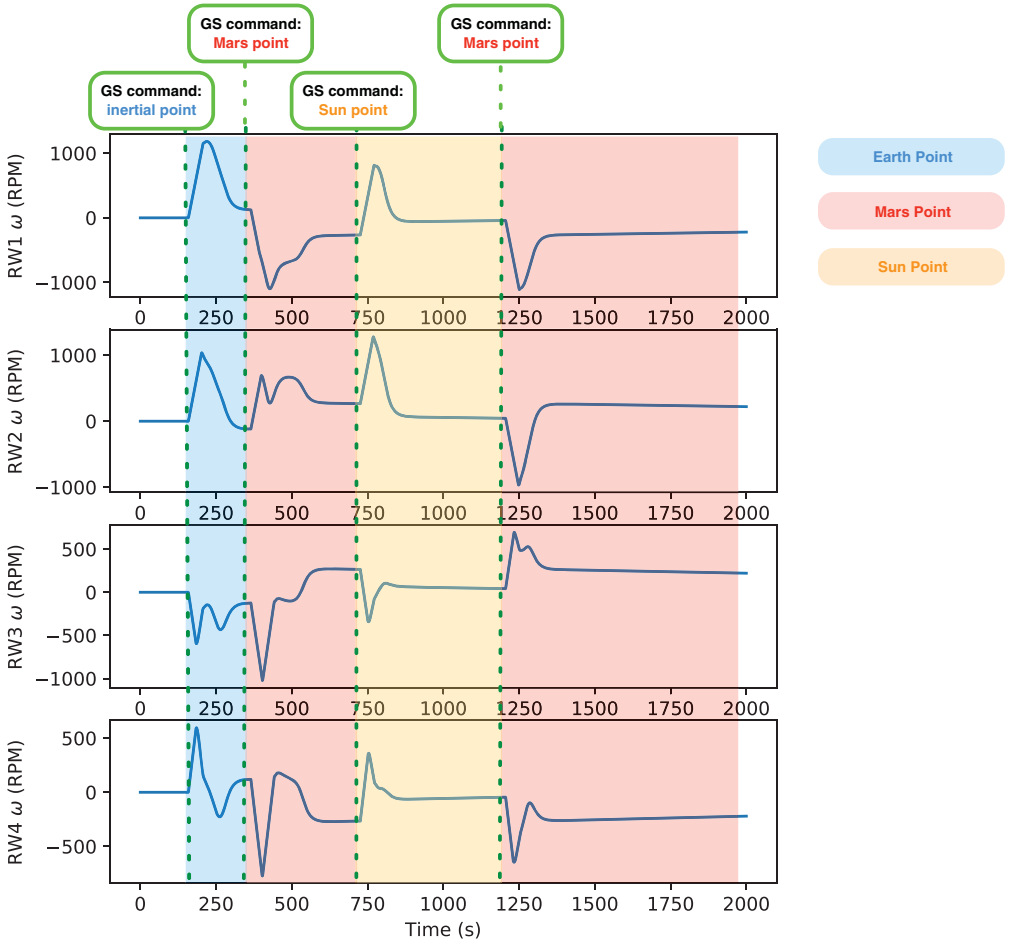
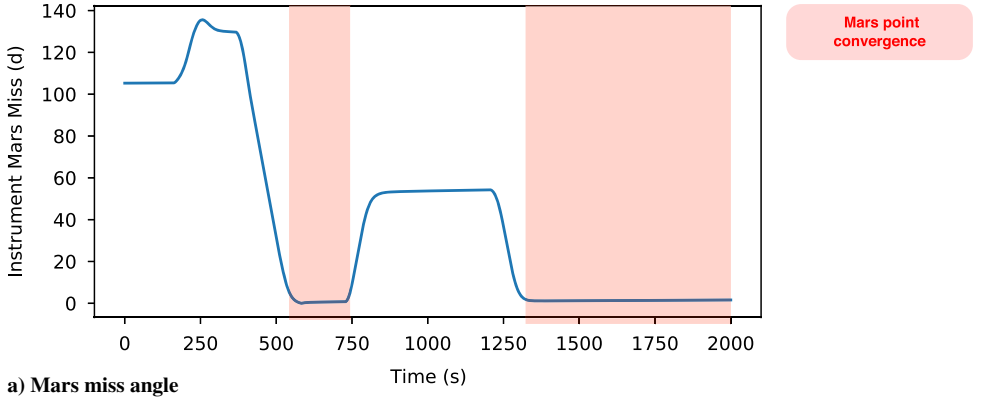
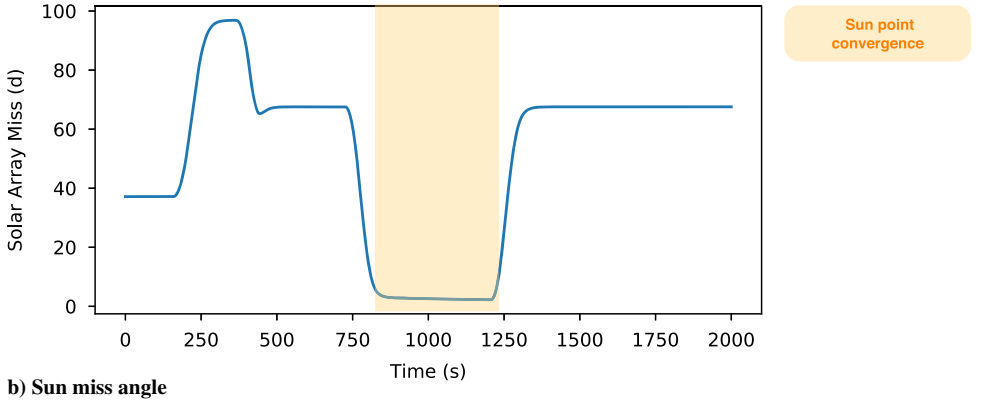


Fig. 11 Closed-loop response: reaction wheel speeds.



a) Mars miss angle



b) Sun miss angle

Fig. 12 Closed-loop response: instruments pointing.

commands. In addition, they constitute a proof of adequate register modeling and exchange of data across BL.

VIII. Results Discussion and Future Work

The previous numerical simulation showcases the use of BL to test nominal functionality of an SC system. Such testing includes not only validating the onboard flight algorithm performance but also verifying commands and interaction with the ground as well as making sure that SC rules are not violated in flight. The scenario shown in this paper involves commanding the SC into several attitude guidance pointing maneuvers. For more complex scenarios, including off-nominal situations in which components fail and sensor signals are corrupted, the reader is pointed to Ref. [15].

Although the feasibility of the BL architecture has been demonstrated, there are several limitations and potential improvements that remain as future work. Firstly, it would be interesting for the Central Controller to accept user interaction during run time. This would allow, for instance, to plug new nodes dynamically upon request of the user or to change the communication time step dynamically. Secondly, the communication and synchronization costs of BL have not been benchmarked or thoroughly analyzed yet. The reason for this is that the biggest limitation found on performance was imposed by the processing capability of the host machine: executing all the different nodes simultaneously tends to be very demanding. This problem can be avoided by running the nodes in a truly distributed fashion, but, from a user perspective, this is more inconvenient because each application needs to be launched separately. In addition, TCP communication across different computing platforms is always slower than within the same platform. Hence, there is a tradeoff that must be made on per-case basis. Lastly, the Tick-Tock synchronization mechanism used in BL does not support multiple hard-time constraints imposed by different nodes. Currently, the architecture can only satisfy a single hard-time constraint, which also has to be the slowest one. When there is a hardware component in the loop that runs in real time (which would be a hard-time constraint), the remaining models must run real time or faster. Although this is a limitation that would be interesting to overcome, in an emulated flat-sat like the one described in this paper, the flight processor emulator is the only component imposing a hard-time constraint and therefore Tick-Tock synchronization mechanism can be used.

IX. Conclusions

The present work has covered the basic aspects of BL, a communication architecture that can be configured to provide an integral SW-sim functionality. BL is currently supporting V&V activities for an ongoing interplanetary mission. Yet, what makes the architecture interesting is its flexibility and its scalability. These features are in turn granted by the adoption of modern software tools and techniques. An abstracted communication layer across a diverse set of nodes is achieved by means of a unique central controller and two generic APIs attached to each of the nodes. These APIs are implemented for nodes whose heterogeneity spans from multithreaded versus single-threaded nodes, asynchronous versus synchronous nodes, little-endian versus big endian nodes, as well as a variety of programming languages: Python, C, C++, and C#. BL demonstrates a novel capability in modern SC simulations where modularity and networking are core capabilities from inception.

References

[1] Vandi Verma, C. L., "SSim: NASA Mars Rover Robotics Flight Software Simulation," *IEEE Aerospace Conference*, 2019, pp. 1–11. https://doi.org/10.1109/AERO.2019.8741862

[2] Cuseo, J., "STK/SOLIS and STK/ODySSy Flight Software: Supporting the Entire Spacecraft Lifecycle," *2011 Workshops on Spacecraft Flight Software*, Johns Hopkins Univ. Applied Physics Lab., Laurel, MD, Oct. 2011.

[3] Biesiadecki, J. J., Henriques, D. A., and Jain, A., "A Reusable, Real-Time Spacecraft Dynamics Simulator," *Digital Avionics Systems Conference, 1997. 16th DASC*, Vol. 2, IEEE, New York, 1997, pp. 8.2-8–8.2-14. https://doi.org/10.1109/DASC.1997.637259

[4] Lim, C. S., and Jain, A., "Dshell++: A Component Based, Reusable Space System Simulation Framework," *Proceedings—2009 3rd IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009*, IEEE, New York, 2009, pp. 229–236. https://doi.org/10.1109/SMC–IT.2009.35

[5] Cameron, J., Jain, A., Dan, B., Bailey, E., Balaram, J., Bonfiglio, E., Grip, H., Ivanov, M., and Sklyanskiy, E., "DSENDS: Multi-Mission Flight Dynamics Simulator for NASA Missions," *AIAA Space 2016*, AIAA Paper 2016-5421, 2016, pp. 1–18. https://doi.org/10.2514/6.2016-5421

[6] Zemerick, S. A., Morris, J. R., and Bailey, B. T., "NASA Operational Simulator (NOS) for V and V of Complex Systems," *Modeling and Simulation for Defense Systems and Applications VIII*, Vol. 8752, International Soc. for Optics and Photonics, SPIE, 2013, pp. 38–44. https://doi.org/10.1117/12.2015246

[7] Grubb, M., Morris, J., Zemerick, S., and Lucas, J., "NASA Operational Simulator for Small Satellites (NOS3): Tools for Software-Based Validation and Verification of Small Satellites," *AIAA/USU Conference on Small Satellites*, Aug. 2016.

[8] Alcorn, J., Schaub, H., Piggott, S., and Kubitschek, D., "Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture," *67th International Astronautical Congress*, IAC Paper IAC-16-C1.1.4, Sept. 2016.

[9] Piggott, S., Alcorn, J., Margenet, M. C., Kenneally, P. W., and Schaub, H., "Flight Software Development Through Python," *2016 Workshop on Spacecraft Flight Software*, Jet Propulsion Lab., Pasadena, CA, Dec. 2016.

[10] Cols Margenet, M., Schaub, H., and Piggott, S., "Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms," *International Symposium on Space Flight Dynamics*, June 2017.

[11] Wood, J., Cols-Margenet, M., Kenneally, P., Schaub, H., and Piggott, S., "Flexible Basilisk Astrodynamics Visualization Software Using the Unity Rendering Engine," *AAS Guidance and Control Conference*, AAS Paper 18-093, Feb. 2018.

[12] Schaub, H., and Junkins, J. L., "Stereographic Orientation Parameters for Attitude Dynamics: A Generalization of the Rodrigues Parameters," *Journal of the Astronautical Sciences*, Vol. 44, No. 1, 1996, pp. 1–19.

[13] Marandi, S. R., and Modi, V. J., "A Preferred Coordinate System and the Associated Orientation Representation in Attitude Dynamics," *Acta Astronautica*, Vol. 15, No. 11, 1987, pp. 833–843. https://doi.org/10.1016/0094-5765(87)90038-5

[14] Wiener, T. F., "Theoretical Analysis of Gimballess Inertial Reference Equipment Using Delta-Modulated Instruments," Ph.D. Dissertation, Dept. of Aeronautics and Astronautics, Massachusetts Inst. of Technology, Cambridge, MA, March 1962.

[15] Cols-Margenet, M., "End-to-End Flight Software Development and Testing: Modularity, Transparency and Scalability Across Testbeds," Ph.D. Thesis, Univ. of Colorado, Boulder, CO, Aug. 2020.

G. P. Brat
Associate Editor