(Preprint) AAS 18-094

# BLACK LION: A SOFTWARE SIMULATOR FOR HETEROGENEOUS SPACEFLIGHT AND MISSION COMPONENTS

**Mar Cols Margenet[*], Patrick Kenneally[†], Hanspeter Schaub[‡] and Scott Piggott[§]**

Dynamic analysis is a validation and verification technique that involves testing, through execution or simulation of a developed product, to detect errors by analyzing responses to sets of input data. In a spaceflight context, dynamic testing addresses inspection of the binary image loaded to the spacecraft hardware as well as its behavior in response to dynamic conditions in a spacecraft operational environment. This paper describes the design and implementation of Black Lion, a purely software-based modern test environment, with the aim to perform dynamic analysis of mission spaceflight software. While this initiative is currently driven by a specific interplanetary mission, the testbed under construction is architected to be highly configurable and modular, allowing for heterogenous software components across multiple computing platforms to be integrated into a single simulation. For example, Black Lion provides the capability to start up and run the operational command and telemetry databases, as well as the unmodified flight software executable. Virtual models are used for the ground system, the single board computer and for other required hardware components like sensors, actuators and avionics. High-fidelity dynamic, kinematic and environment models are integrated in order to test the flight software system in realistic closed-loop simulations.

## INTRODUCTION

Black Lion (BL) is a communication architecture for what is known as a distributed software-simulation (SW-sim). While 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, the BL architecture is general enough to create a range of distributed spacecraft simulations. Multiple software processes across heterogenous computing platforms can be integrated into a single BL simulation. This could allow, for example, having a dedicated computer running a complex space environment model and another computer integrating spacecraft dynamics, both of them exchanging data dynamically through BL. This paper focus specifically in the use of BL in a software-based flat-sat configuration.

Considering the flat-sat testing 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 spacecraft (SC) models to simulate the dynamics, kinematics and environment (DKE). In contrast, SW-sim testing uses virtual models or emulators in place of hardware components such as the GS or SBC. The *raison d' être* of a 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 spacecraft's DKE. As illustrated, the GS emulator ingests the same mission scripts as the real ground system and contains also the same command/telemetry databases, while the SBC emulator runs the actual mission FSW.

[*]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.
[†]Graduate Student, Aerospace Engineering Sciences, University of Colorado Boulder.
[‡]Professor, Glenn L. Murphy Chair, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431. AAS Fellow.
[§]ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics, University of Colorado Boulder.

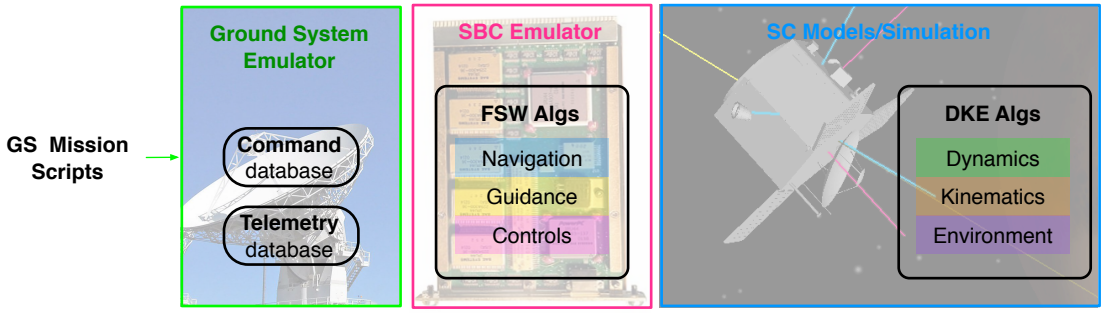


**Figure 1. Virtualization of Spaceflight Components.**

The virtual models used in a SW-sim 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 (written in different programming languages, nominally running at different speeds, etc.). Therefore, a common communication architecture is needed to guarantee a computationally performant and synchronized exchange of data between the multiple nodes (virtual components).

Black Lion is being built to provide this communication architecture to connect all the 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 virtual models. While the Black Lion effort is currently motivated for a specific spacecraft interplanetary mission, the system is being built under the principles of reusability and scalability.

The main purpose of a SW-sim is to test the onboard FSW executable, a complete binary image that includes the FSW algorithms/application, the real-time operation system (RTOS), the Boot SW and the support package for the single board computer as illustrated in Fig. 2. Testing the onboard FSW in response to dynamic stimuli in a spacecraft operational environment is also one of the driving goals for flat-sat testing. While pure SW-sim testing does not replace flat-sat tests, 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 by using software models only, providing a flexible and cost-effective means of performing mission system testing early on in a mission program's schedule.

Pure SW-sim environments have long been used in the aerospace industry. Ongoing aerospace missions using a software-only test based approach include James Webb Space Telescope, Space Launch System, Juno and OSIRIS-Rex among others. There exist different flavors of SW-sim architectures: some groups are developing their own in-house solutions (like the Black Lion SW-sim) while other groups may have

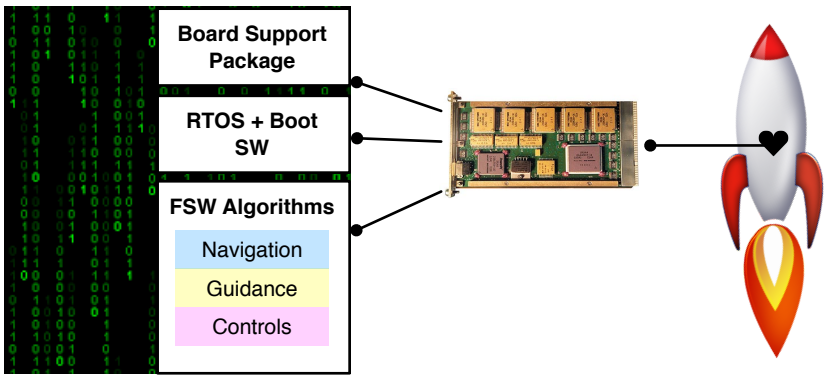


**Figure 2. Binary Image Loaded to the Spacecraft (Onboard Flight Software).**

acquired and maintain versions of Lockheed Martin's "SoftSim" to support verification and validation (V&V) activities. 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-sim's are meant to provide a common functionality: the ability to run the unmodified FSW executable in a software-only simulation environment.

The BL SW-sim in particular provides the capability to start up and run the operational command and telemetry databases, as well as the unmodified flight software executable. Virtual models are used for the ground system, the single board computer and for other required hardware components like sensors, actuators and avionics. High-fidelity dynamic, kinematic and environment models are integrated in order to test the flight software system in realistic closed-loop simulations. The scope and coverage of the BL simulator involves assuring that the source code components can reliably perform required capabilities under both nominal and off-nominal (i.e. faulted) conditions and that the system's responses are deterministic.

Whereas the functionality and coverage of the BL SW-sim are equivalent to those pursued by other groups, the novelty of the BL approach lies in the adoption of modern software development technologies and techniques. The idea is to move away from highly expensive, specialized products and focus on delivering scalable functionality and increased operational efficiency at a faster pace and lower costs. In order to achieve this modern shift, the following directives are driving the system design and implementation: 1) Use of open-source, cross-platform products, 2) Modular architecture by initial design and 3) Agile software development.

The paper is outlined as follows: first, there is a preamble introducing the FSW application that is the central target of the BL SW-sim testing. Next, the different components to be included in the SW-sim are defined. The following section addresses communication aspects between the different nodes, covering: node heterogeneity, communication goals and introduction the BL architecture. Then, there is a special section focusing on the adoption of modern SW technology and its role in the BL development effort. The next section explains the strategies for both the data transport (socket patterns, connections, etc.) and node synchronization. A brief section then highlights the capability of BL to run as a distributed machine system. Finally, some conclusion remarks and a roadmap for future work are included.

## PREAMBLE: THE FSW ALGORITHMS AND APPLICATION DEVELOPMENT

The collaboration between LASP and the AVS laboratory encompasses developing GN&C algorithms for different mission profiles. In order to develop these algorithms a parallel development effort has produced a general SW astrodynamics framework aimed at serving as a testbed for prototyping and testing. This SW astrodynamics framework is named Basilisk (BSK) and is currently available as an open-source product.[1] BSK leverages Python's ease-of-use as a testbed for FSW development provided that the spacecraft 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 Basilisk software framework is depicted in Fig. 3. As illustrated, the system is decomposed into two main blocks: a high-fidelity simulation of the physical spacecraft ("SC Models" in Fig. 3) and a GN&C algorithms suite ("FSW Algs." in Fig. 3). Both the SC simulation and FSW processes 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. 3). In the general case, the FSW algorithms are written exclusively in C as per requirements of spacecraft missions.

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 an actual flight target. The flight target could be either a specific processor and RTOS or a middleware layer like NASA's core Flight System (cFS). Targeting middleware is advantageous in that ensures portability among different processors and RTOS. Reference 2 discusses the ease of transition from the Basilisk desktop environment into a cFS application, ready to be embedded onto an SBC running a standard RTOS. Figure 4 showcases a system where the FSW algorithms are integrated within cFS, which in turn run on an RTOS inside an SBC emulator. A virtual SBC would be used in a SW-sim, while a physical SBC would be used in a flat-sat system. Figure 4 illustrates that the communication between the two separate systems happens through a TCP connection, allowing the different components to run on different computers. While References 2, 3 showcase specific strategies to enable peer-to-peer com-

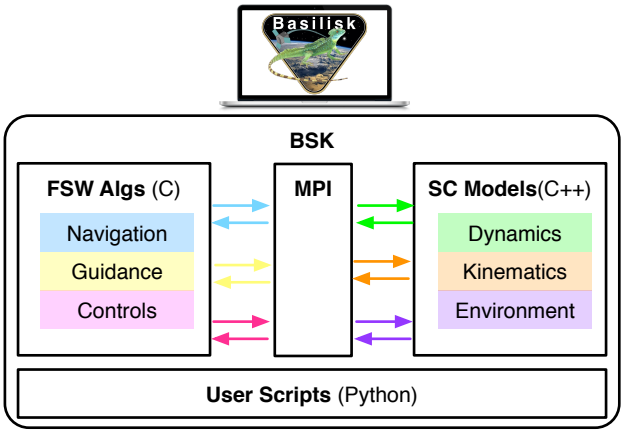


**Figure 3. Spacecraft Models and FSW Application Developed within Basilisk (BSK).**

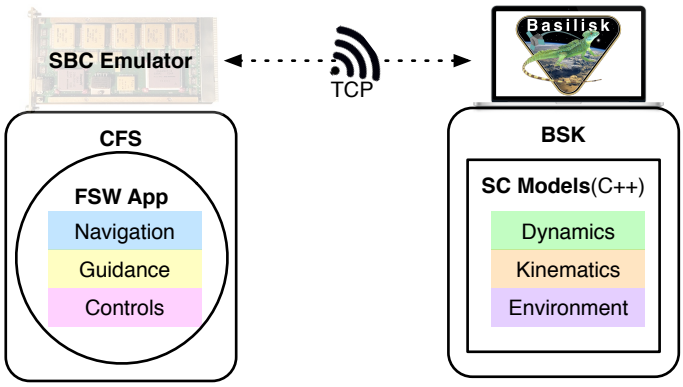


**Figure 4. Spacecraft Models in BSK and FSW Application Migrated to a Flight Target.**

munication between BSK SC models and a given FSW application, the BL SW-sim can now be used in a much more flexible way to allow broadcasting communication between an indefinite number of peers. The remaining sections of this paper focus on the functionality and application of BL.

## EXPANSION TO MULTIPLE NODES

Once the GN&C flight software application is determined, the next step is to expand the software simulation coverage by simulating multiple spaceflight components in order to allow simultaneous and rapid iteration testing from multiple engineering groups. The right hand side of Fig. 5 illustrates all the different components that are to be included into the distributed BL SW-sim. As depicted in Fig. 5, the initial system with only the FSW application and the SC models is expanded to include: a GS virtual model, an SBC virtual model and a visualization tool.

**Ground System Emulator:** 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 CCSDS packets (the CCSDS File Delivery Protocol is a file transfer protocol used in space applications). An example of GS emulator is the open-source COSMOS.*

**Virtualized Single Board Computer:** It contains the unmodified FSW executable for the mission, which runs on a standard RTOS. The SBC emulation includes also FPGA-like registers. In this case, the

*http://cosmos-project.org

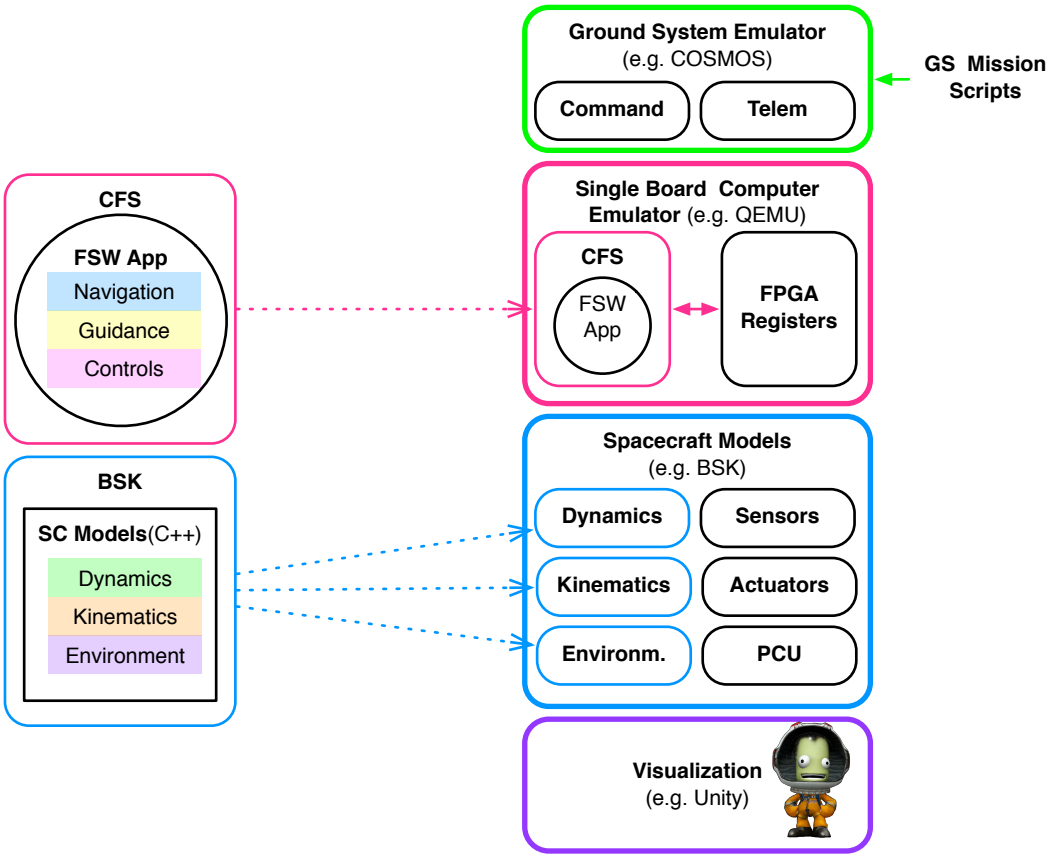


**Figure 5. Expansion from GN&C Testing to an Integral SW-Sim Testing Environment.**

actual FPGA registers are emulated/replaced with a memory map for input/output of raw binary data. An example of SBC 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 high-fidelity DKE models on one side and hardware component models on the other side. The hardware component models include sensors (gyro, star tracker, coarse sun sensors, etc.), actuators (reaction wheels, attitude control thrusters) and the power control unit (PCU). Jointly, the DKE and hardware models allow testing of other nodes in closed-loop dynamics simulation.

**Visualization:** a graphical user interface (GUI) is being developed with the Unity‡ game engine. The GUI is meant to reproduce the spacecraft physical behaviors, which in this case are determined by the BSK simulation. Playback and speed up/down reproduction are capabilities included in the visualization tool. Reference 4 describes the GUI under development that is going to accompany BSK.

## COMMUNICATION BETWEEN THE COMPONENTS

Recall that the nodes in the SW-sim are stand-alone applications, which are initially unaware of any other nodes. They are written in different programming languages, wrap their internal data using different structures or packet types, and run at different speeds.

*http://qemu.org
†http://hanspeterschaub.info/bskMain.html
‡http://unity3d.com

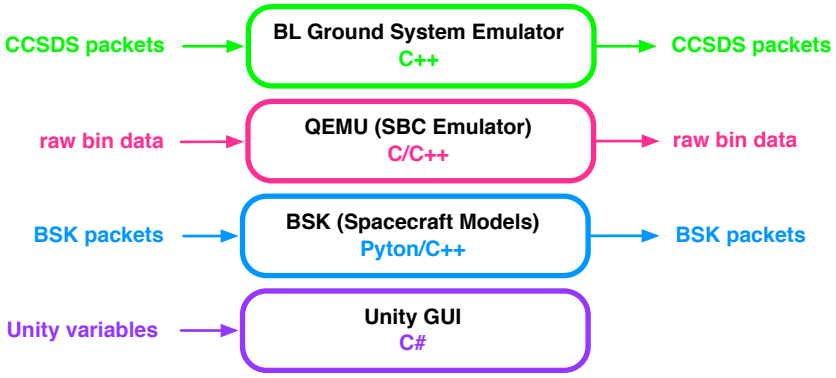


**Figure 6. Heterogeneity of Programming Languages and Internal Data Packets in the BL SW-Sim.**

The differences between the particular nodes currently used in BL are highlighted in Fig. 6. As a quick recapitulation, the GS modelled is written in C++ that uses CCSDS packets with a particular data format. The SBC emulator is based on the open-source product named QEMU. It is written in a combination of C and C++, and deals directly with raw binary data. The SC models are simulated within BSK. While 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 that come along with a message header. Finally, the Unity-based GUI is written in C#.

The heterogeneity between the multiple components drives the need of a dedicated architecture in order to achieve communication. The term communication, as understood here, involves multiple goals: **1) Transport** of binary data (share bytes between nodes), **2) Serialization** of binary data (each node must know how to convert the received bytes into structures that can then manage internally), **3) Synchornization** of nodes (all the nodes must be kept in lock-step during the simulation run), **4) Dynamicity** in the connections map (the less static, or strictly required, pieces there are in the communication network, the better).

The goal of BL is to achieve the described communication goals while being completely transparent to the internals of each node such that users can plug and play with their models of choice, while having the flexibility to add/remove nodes because they are not static pieces in the communication map. To this end, the BL architecture depicted in Fig. 7 is comprised of a central controller and two APIs that are attached to each node.

**BL Central Controller:** is the one and only static piece in the network (i.e. it has a static IP address). The central controller acts as a master in the synchronization of nodes and as a broker in the transactions (exchange of data) between nodes.

**Delegate API:** It manages sockets and direct connections with the central controller. It is the same script attached to all the nodes. The `Delegate` class is currently implemented for Python nodes and for C++ nodes.

**Router API:** It is a generic class with node-specific callbacks. 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 system format, and then passes the data to the node's Delegate API, who is ready to ship it across.

In order to explain better the idea behind the given architecture, one can use a human language analogy: each node is an individual that speaks a different language, as illustrated in the right hand side of Fig. 7. The router acts as a translator from the individual's language to a common standardized language, like English in the case of Fig. 7. Once the Router has translated the data, the Delegate communicates over 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.

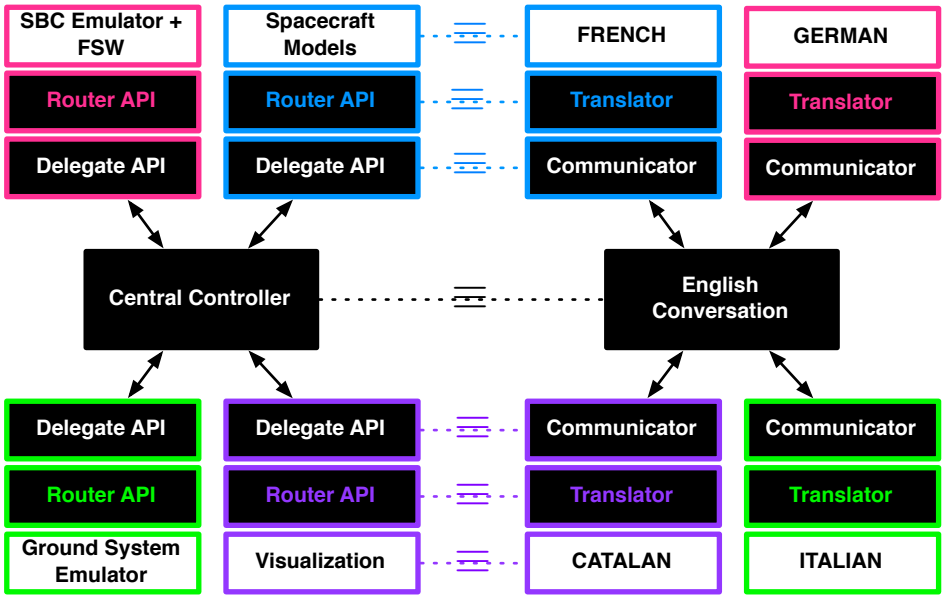


**Figure 7. Communication Architecture: Central Controller, Delegate APIs and Router APIs.**

## ADOPTION OF MODERN SOFTWARE TECHNOLOGY AND TECHNIQUES

Before moving into the details of the communication hub, it is interesting to step back and emphasize the modern technology and techniques that are critical to the BL effort. BL takes advantage of the following open-source, cutting-edge technologies:

**ZeroMQ Message Library:** * 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 range of programming languages, which can perfectly interact with each other.

**Google Protobuffers:** † 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.

In terms of techniques, the BL software is being developed in a **component-based** (modular) approach where nodes are gradually integrated one by one. Further, it is very easy to swap pieces, add new ones, or remove existing ones to perform different kinds of testing. This gradual escalation is granted by having a **dynamic architecture**, where the central controller is the one and only static piece (i.e. server) and the nodes are dynamic clients that can come and go on the fly. Moreover, the software is built through what is known as **agile development**, which implies continuous delivery to mission users and immediate integration of the feedback received - resulting in very fast build, testing and deployment cycles.

## DATA TRANSFER AND SYNCHRONIZATION

In order to understand how the communications hub works, it is critical to explain upfront the socket types and connection types used in the system. Two types of ZMQ-socket patterns are used to transport data: the request-reply pattern and the publish-subscribe pattern. The publish-subscribe pattern is applied in two different flavors, as described next:

*http://zeromq.org
†http://developers.google.com/protocol-buffers

**Request (REQ) - Reply (REP):** the central controller has a REQ socket for each node instantiated in the simulation that is used to make requests. In turn, each node has a REP socket that receives and parses the request, does somethings with it, and eventually replies back indicating accomplishment.

**Publish (PUB) - Subscribe (FRONTEND SUB):** Every node has a PUB socket to share its own internal data by publication. 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):** The central controller has, additionally to a SUB-frontend, a PUB-backend. The messages received in the frontend are internally routed to the backend, which then re-publishes 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 just described is exemplified in Fig.8. The figure depicts the central controller in the middle and two sample nodes highlighted in blue and magenta respectively. As shown in Fig. 8, the sockets are encapsulated by the Delegate API.

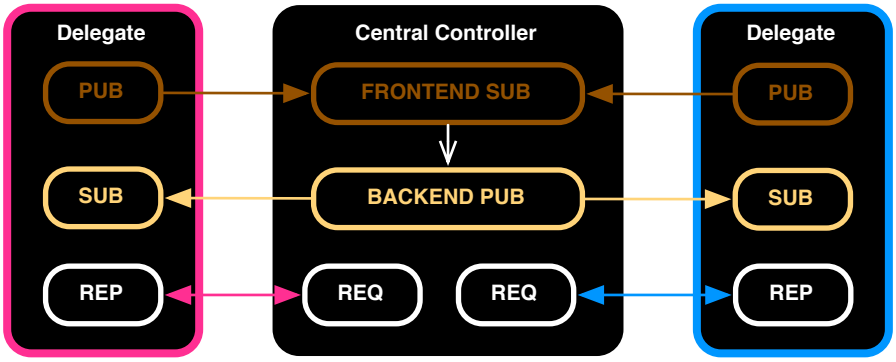


**Figure 8. Socket Patterns between the Central Controller and Sample Nodes.**

Now that the socket types are clear, let us overview the connections of these sockets to a given IP address and port. All the socket connections in the system fall into either one of these categories: static connection (i.e. **binding** type in ZMQ terms) or dynamic type (i.e. **connecting** type in ZMQ terms). The static connections are all associated to sockets in the central controller, while the dynamic connections are associated to the sockets in each of the nodes' Delegate API.

**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 an 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 server is always required and the clients are independent entities that do not intrinsically rely on each other. The use of ZMQ also allows all the connections to be protocol independent (TCP, IPC...). The idea of socket binding (static nature) versus socket connecting (dynamic nature) is illustrated in the topology showcased in Fig. 9. The figure also reflects the fact that there is only one IP address in the entire BL system and within this address multiple ports are used. As before, the figure displays the central controller (server) in the middle and two sample nodes (clients) highlighted in blue and magenta colors.

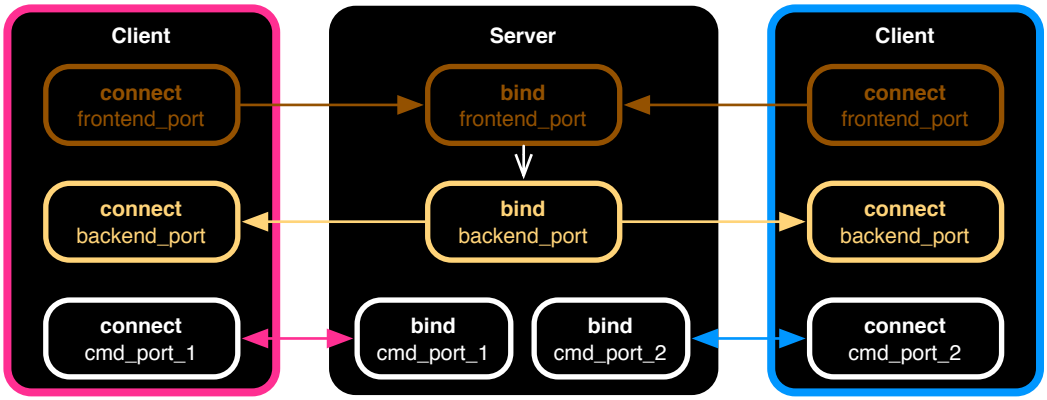


**Figure 9. Socket Connections Types (Binding vs. Connecting) and Ports.**

### Request-Reply Communication between the Controller and the Nodes' Delegate

The requests from the Controller to the Delegate on each node are not spacecraft commands, they are communication and synchronization commands exclusive to the SW-sim. In the current BL implementation, the controller can make 5 type of requests, some of which come in the form of multi-part messages. Firstly, there is the **"Initialize" request**, which is a multi-part message containing the "Initialize" signal, the controller's frontend address and port and the controller's backend address and port. The actions taken by the node when this request is parsed are: self initialization, connect its pub-socket to the controller's frontend and connect its sub-socket to the controller's backend. Secondly, there is the **"Provide Desired Message Names" request**, which instructs each node to report all the message names to which the node wishes to subscribe. Thirdly, there is the **"Match Message Names" request**, which is multi-part message with the "Match" signal and a set of all the message names that the other nodes have asked for. The node returns back a reduced list with the message names for which it has found an internal match. Then, there is a **"Tick" request**, which is used at every time-step of the SW-sim run for synchronization purposes and it contains the time duration of the next time-step (i.e. $\Delta t$). Once the node has accomplished all the tasks, it sends back a "Tock" reply. Eventually, there is **"Finish" request**, which is a signal for the node to close the sockets, clean up and shut down.

### Tick-Tock Synchronization

Three actions or tasks happen in sequence inside each node between the parsing of a "Tick"-request and the sending of a "Tock"-reply: **publish**, **subscribe** and **step simulation** forward. Note that these three actions are node internal calls triggered by the "Tick" request. Figure 10 depicts the sequence of interactions and actions happening between the central controller and a sample node highlighted in magenta. In Fig. 10, the words written in white imply interactions between the controller and the node whereas the words written in magenta indicate node internal calls.

**Publish:** in the publish internal call, the node's Router is responsible to collect the application internal data and make it available to the node's Delegate, who sends it out as publications that the controller's frontend will grab.

**Subscribe:** in the subscribe internal call, the node's Delegate receives external data coming from the controller's backend and hands the data to the node's Router, who is responsible to write these messages down into the internals of the node application.

**Stem Simulation:** in general terms, the step simulation internal call implies executing the node's application during $\Delta t$ in order to generate new data. Recall that $\Delta t$ is a message part of the "Tick" request sent by the controller. Having said that, there are different nuances in the precise meaning of "step simulation" for nodes that are synchronous (i.e. run in cycles, like FSW or the spacecraft simulation) and for nodes that are asynchronous (i.e. are event-based, like the ground system)

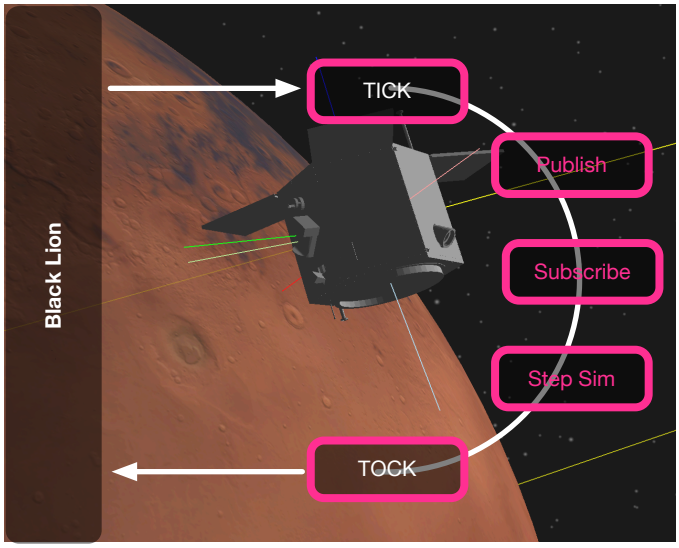


**Figure 10. Node Actions between a "Tick-Tock": Publish, Subscribe, Step Simulation.**

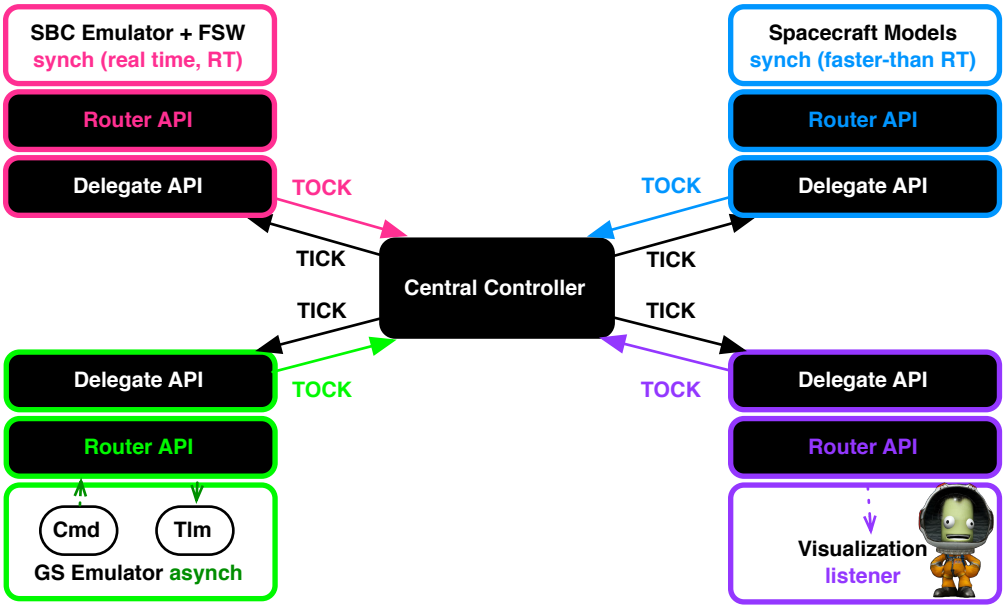


**Figure 11. Nodes' Timely Nature: Syncrhonous, Asynchronous and Listener Behaviors.**

Because each node is an independent process that runs at a different speed, the "Tick-Tock" signal ensures that all of them are in lock-step. Let us recall the particular four nodes that are being integrated in the BL system: the SBC emulator, the SC simulation, the GS emulator and the visualization GUI. Figure 11 depicts the synchronous nature of the SBC and SC simulation nodes, the asynchronous nature of the GS emulator, and the listener nature of the visualization node.

Both the SBC node and the SC simulation node are synchronous in nature in the sense that they run in cycles or at predefined rates. Because the FSW executable is running inside the SBC emulator using a RTOS, the speed of the FSW execution is real time. In contrast, the SC simulation runs natively faster than real time. For the synchronous nodes, the "step simulation" call implies running as many cycles as there are within $\Delta t$ before exchanging data again with the rest of the system. If one node finishes simulating $\Delta t$ earlier, 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. In contrast, the GS node is asynchronous: the sending of spacecraft commands and the receiving of telemetry are discrete-time events. The GS also receives a "Tick" command because the exchange of data still happens simultaneously between all the nodes. The asynchronous nature of a node, like the GS, demands a special treatment of the Delegate and Router APIs: the communication through the APIs must happen at a different thread from which the main application is running. The Visualization node is another case on its own: it can be simply regarded as a "listener" governed by the SC simulation. Therefore, it only subscribes to the SC simulation messages and, within the "step simulation" call, its job is to show the spacecraft timely evolution according to the received set of messages.

## MULTI-MACHINE FUNCTIONALITY

On a final note, BL is capable of running as a distributed system architecture with multiple machines, different operation systems, talking to each other. Interestingly, this multi-machine functionality has came out-of-the-box thanks to using modern SW technplogy such as ZMQ. In fact, the multi-machine system in Fig. 12 depicts the architecture with which BL usually runs for development and testing at the present. It is *da facto* that the BL SW-sim can perfectly run all the nodes within the same computer, provided that the computer has enough capacity to handle all the concurrent applications.

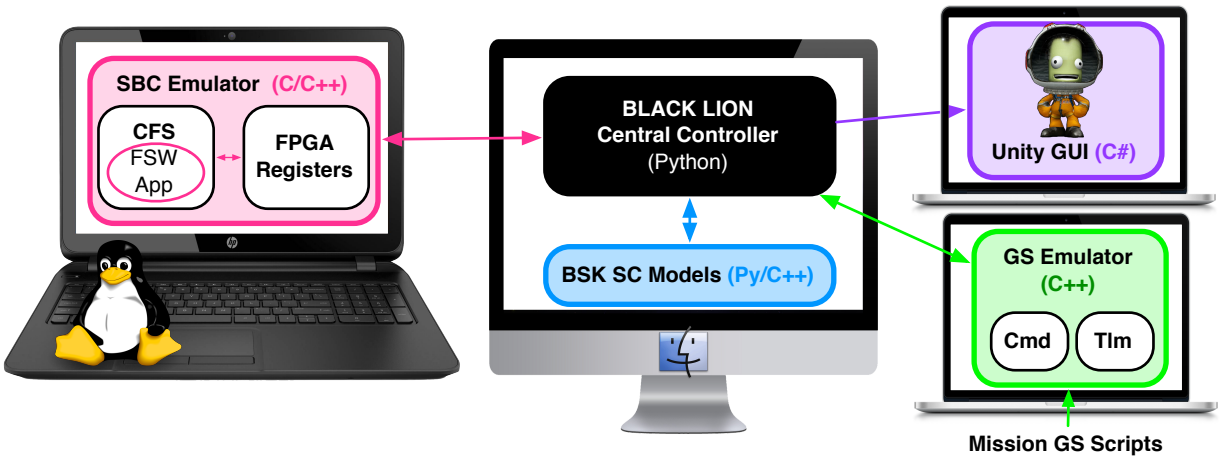


**Figure 12. Out-of-the-box Multi-Machine Functionality.**

## CONCLUSIONS AND FUTURE WORK

The present work has covered the basic aspects of Black Lion, a communication architecture that can be configured to provide an integral SW-sim functionality. BL is currently supporting validation and verification 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 diversity of nodes is achieved by means of a unique central controller and two generic APIs attached to each of the nodes. Currently, these APIs have been implemented for nodes whose heterogeneity spans from: multithreaded vs. single-threaded nodes, asynchronous vs. synchronous nodes, as well as for a variety of programming languages: Python, C and C++. The C# counterpart of the APIs is currently under development. Other tested features of BL are cross-platform compatibility and distributed system (i.e. multi-machine) capabilities. Ongoing and future work encompasses:

**Dynamic discovery of nodes:** implies 1) Configuring nodes automatically to enter into a particular mode or scenario and 2) Launch/start-up nodes remotely from a single common call within the Black Lion central controller.

**Graceful handling of node failure:** if one node fails, the user can decide whether to continue running the SW-sim test with the remaining nodes or to restart the simulation run. Further, description of failure modes and recovery documents are *da facto*.

**Direct pipe for fault injection:** corruption of spacecraft models (sensors and actuators) and corruption of CCSDS packets can be triggered dynamically from the central controller.

**Command and telemetry checking routines:** implies 1) Integration of test registrations to see flow of commands and telemetry during run time and 2) Addition of simple callbacks to telemetry checks (reporting passed/failed states) for runs in which test registrations are not included.

**Single-step execution and halt command:** the tester can place halt commands within the central configuration script in order to run the SW-sim to a specific point and then poke around in the sense of, for example: reading out memory locations, inserting new values into memory locations, examining stacks, attaching debuggers to various processes, and so on.

**Unification/definition of data interfaces between applications:** Google protobuffers are to be implemented instead of Interface Control Documents (ICD) or Electronic Datasheets (EDS). The use of protobuffers enables backwards compatibility regarding changes on the overlay data definition. Without backwards compatibility, changes on data structures (i.e. change of versions) may result in insidious errors, because the most efficient way to pass data is binary format and mismatches in format/versions can go undetected until much later and so can be extremely hard to find.

**Handling different SW-sim build configurations:** implies handling of 1) Nodes built in 32 bits and others in 64 bits (note that some data sizes change when these different options are used, which can mess up the access of the defined data structures), 2) Teams using different compilers (there can be very subtle differences in data representations) and 3) Nodes with different endianness.

## REFERENCES


[1] J. Alcorn, H. Schaub, S. Piggott, and D. Kubitschek, "Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture," *67th International Astronautical Congress*, Guadalajara, Mexico, Sept. 26–30 2016.

[2] S. Piggott, J. Alcorn, M. C. Margenet, P. W. Kenneally, and H. Schaub, "Flight Software Development Through Python," *2016 Workshop on Spacecraft Flight Software*, JPL, California, Dec. 13–15 2016.

[3] M. Cols Margenet, H. Schaub, and S. Piggott, "Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms," *International Symposium on Space Flight Dynamics*, Matsuyama-Ehime, Japan, June 3–9 2017.

[4] J. Wood, M. Cols-Margenet, P. Kenneally, H. Schaub, and S. Piggott, "Flexible Basilisk Astrodynamics Visualization Software Using the Unity Rendering Engine," *AAS Guidance and Control Conference*, Breckenridge, CO, February 2–7 2018.