(Preprint) AAS 20-134

A NEW MESSAGING SYSTEM FOR BASILISK

Scott J.K. Carnahan[*], Scott Piggott[†], Hanspeter Schaub[‡]

The Basilisk Astrodynamics Framework utilizes a messaging system to specify and implement interfaces between spacecraft simulation modules. A new messaging system has been developed for the Basilisk Astrodynamics Framework to enable or better enable multiple-spacecraft simulations, multi-threaded simulations, dynamically allocated message payloads, message connection by users, and message type-checking. Templated C++ functor classes are used for message read and write operations, providing direct but controlled access to message memory. Memory-safe and bit-for-bit repeatable multi-threaded simulations are also enabled by the functor implementation.

INTRODUCTION

Companies from OneWeb[*] to Amazon[†] have already begun launching or will launch thousands of satellites into orbit in the coming years. With these plans comes the need to simulate multiple space systems simultaneously. Of the major astrodynamics software packages available, most allow for the simulation of multiple spacecraft, but none apparently do so in a multi-threaded manner that takes advantages of modern multi-core CPU architectures[‡]. Aside from the obvious impediment to simulating thousands of spacecraft (compute time), other challenges face the engineer attempting to analyze such a constellation. These challenges are largely associated with the book-keeping of inputs and simulation initalization.

In the original Basilisk messaging system (seen in Fig. 2), messages were assigned specific names (string objects). Messages could be referenced by any module by name. This original messaging implementation had the benefit that it allowed for strict access to control to the message content. Further, the messaging setup created an application programing interface which facilitated a module simulation framework. Particular components, such as an attitude estimator,[1,2] can readily be exchanged without having to recompile the software. Rather, the new module is connected on the outer Python layer. However, this system can be cumbersome and error prone when there are dozens or hundreds of nearly identical but unique messages to name (due to modeling multiple spacecraft). Therefore it is desirable to make messages identifiable without the need for users to define evermore unique message names. Further, the speed of accessing these messages and logging them has been identified as a bottle neck in trying to improve overall evaluation times.

[*]Engineer, Cesium, 13412 Galleria Circle, Suite H-100, Austin, TX, 78738

[†]ADCS Lead, LASP, 1234 Discovery Drive

[‡]Professor and Glenn L. Murphey Endowed Chair, Aerospace Engineering Sciences, CU Boulder, 3775 Discovery Dr, Boulder, CO 80303

[*]oneweb.world

[†]https://www.geekwire.com/2019/amazon-project-kuiper-broadband-satellite/

[‡]The Formation Algorithms Simulation Testbed at JPL utilizes DARTS and is not generally available, but does allow for multiple spacecraft dynamics to be calculated on separate CPUs entirely.

Templated C++ message classes and associated read and write functors simultaneously provide the necessary solutions for user interface and multi-threaded applications. The key for user interface and unique message identification is that messages become public class attributes for classes that write them. Therefore they can be connected to directly in Python. The read and write functors allow modules interfacing with a message to have direct and controlled access to the message data without tracking a message name or ID. Finally, the "statefulness" of functors provides the necessary memory management capability to safely provide multi-threaded speed to the Basilisk Framework. FIGURE illustrates the new messaging system.

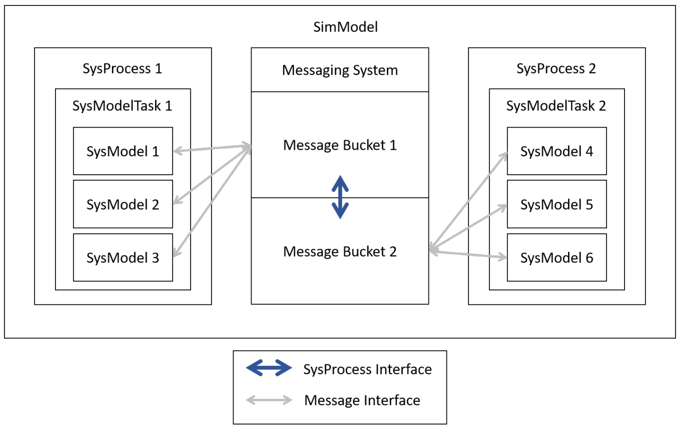

Figure 1. In the original system, messages were exchanged betwen modules and the messaging system storage (small arrows). Messages could be exchanged across SysProcess boundaries via process interfaces (large arrows).

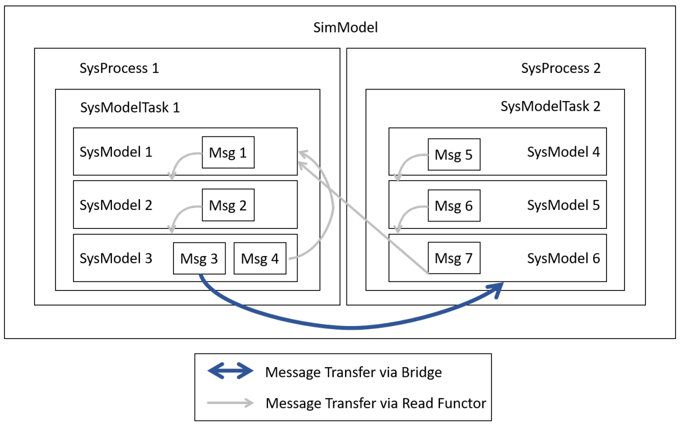

Figure 2. In the new system, messages are class attributes that can only be read by Read Functor classes of the correct type (small arrows). Read Functors are also typically members of the classes that need the message, although that is not shown here. An envisioned Bridge class (large area) would be instantiated by the user and allow single-message transfer across SysProcess boundaries.

THE ORIGINAL SYSTEM

Basilisk[*] is one of a variety of software available to astrodynamics users. Other well known products include GMAT[†], STK[‡], Trick[§], FreeFlyer[¶], and Dshell++.[3] Basilisk also lives within an ecosystem of tools aimed at more general simulation of robotics and dynamics including JPL's DARTS [‖] and the Robotic Operating System (ROS)[**]. These in turn are part of a larger group of simulation software generally. Many large simulation software packages today utilize the concept of modularization. In these packages, modules of code that perform distinct functions are developed in such a way that they are independent of the execution or even the presence of one another. These modules are put together to form a larger, more complex simulation. In order to do this, they must share data in some way. The most useful methods of sharing this data will follow some standard coding guidelines. For instance, if the software is object oriented, the method of sharing information ought to emphasize encapsulation, single purpose, and "Don't Repeat Yourself" (DRY), among others. In scientific and high performance computing, this is often done over a Message Passing Interface (MPI) [††]. MPI specifically allows for information sharing across compute processes and even across discrete hardware. ROS, for instance implements a message passing interface.

Basilisk does not itself utilize a message passing interface.[4] However, Basilisk companion software like Black Lion and Vizard do.[5] Instead, Basilisk implements a "messaging system". This system borrows from methods and concepts within MPI, but implements them in a way which is not focused on cross-hardware and cross-process computing. The goal is to allow modularization and loose coupling of code modules in a single simulation. As currently found on the Basilisk master branch ("the original system" for the remainder of this paper), the messaging system uses a publish/subscribe model where in messages are given string-type names. Other modules can subscribe to those messages if they are provided with that string-type name. No connections are actually made when the user sets up a simulation. Rather, the messaging system is called for each module in a series of self-initialization and cross-initialization calls when a simulation is initialized. These steps actually form the connections between modules based on integer-type message IDs which are paired with message names and exchanged internally and out of view of the user. All messages are handled (subscribed, written, and read) by a singleton messaging system class which stores messages in a central memory location per task group. Messages have associated meta data that tracks storage location, publisher and subscriber information, and read and write history. Additionally, users are prevented from reading and writing messages across task group boundaries unless such a capability is specifically requested between two distinct task groups. The user is then allowed to read and write any messages across that boundary.

Pros of the original system

There are several positive attributes of the current messaging system. The first is that the original Basilisk messaging system accomplishes the goals which it set out to achieve. Namely, it allows for decoupling of models in the framework and the building of complex models out of basic engineering

[*]http://hanspeterschaub.info/basilisk/index.html
[†]https://software.nasa.gov/software/GSC-16228-1
[‡]https://licensing.agi.com/stk/
[§]https://github.com/nasa/trick
[¶]https://ai-solutions.com/freeflyer/
[‖]https://dartslab.jpl.nasa.gov/DARTS/index.php
[**]ros.org
[††]https://www.open-mpi.org/

building blocks. Moreover, it accomplishes this efficiently in that Basilisk is highly performant even with this model decoupling which necessarily adds compute overhead to any simulation in order to transfer data from one model to another. Furthermore, the Basilisk messaging system is simple to use as users need only to name a message with a human-readable string-type and provide the same string to any model that needs to read that message. Finally, the implicit connection of modules via the original messaging system is advantageous in the flexibility it allows during simulation set up. Specifically, modules can be provided a string-type name when they are instantiated whether or not that message has already been instantiated in the simulation.

To summarize the advantages of the original Basilisk messaging system:

- Loose coupling of simulation models
- speed of connections
- simplicity and readability of message identifies to users
- implicit message connections

Cons of the original system

On the contrary, the original Basilisk messaging system has negative attributes as well. Some of them are the same attributes as the positives listed above, but viewed from a different angle. First, although the system is efficient, it has areas where it could improve its speed. Notably, the user interface to the messaging logs (time histories of messages) can consume a significant percentage of the simulation time when large amounts of data are recorded and plotted.

Next, the simplicity and readability of string-type message identification can also be error prone. Users can easily make typos which are not caught by Python's run time error checking, the IDE, or the messaging system itself. The messaging system is unaware if "reaction_wheel_speed" is a typo or intentionally distinct from "reaction_wheel_speeds". Therefore, there is no clear indication to the user that these mistakes have been made. To continue that point, message logs are obtained by strings which are then executed by Python to pull the data and the strings are prone to errors in this case as well.

Another con of the original messaging system is the implicit connections. Although it is convenient in simulation set up because it does not require specific ordering, there are multiple problems with this. First, implicit connections as implemented are not type-safe. This means that models which need one message type can subscribe to a different message type without throwing an error. Occasionally this feature is so nefarious as to produce a working simulation even with improper message connections. Second, because messages do not have to exist in order to be subscribed to, users may (via typo, for instance) subscribe to a message that does not exist. Third, implicit messaging may be hidden from the users, obfuscating the mechanics of a simulation. This mainly happens due to default message names. For example, the SpacecraftPlus model has a default dynamic state message name of "inertial_state_output" and various models that need this message default their input to "inertial_state_output". This connection is made without and perhaps in spite of user intention or knowledge.

A smaller con of the original messaging system is the use of message types to find and access messages. All messages in Basilisk are custom struct typedefs. From the Python level, these types are searched for in the message data to pull message logs.

Finally, the original Basilisk messaging system was written with a focus on models written in C for flight software applications. This limits the types of data that can be contained in a message and also leads to many lines of conversions between types of variables that are useful in C++ and those that are acceptable in C. Finally, the singleton design pattern of the original messaging system prevents multi-threading Basilisk at the task or task group level.

To summarize the disadvantages of the original Basilisk messaging system:

- Interface and speed of user access to message logs
- Error propensity and weak error handling due to string-type message identifiers
- Weak message typing
- Implicit connections
- Reference by message type
- C types only
- Inherently Single-threaded

THE NEW SYSTEM

This paper details a new messaging system for Basilisk that solves the disadvantages of the original messaging system in the following ways:

- Disadvantage: Interface and speed of user access to message logs

  Solution: Messages and message logs are conceptualized and implemented as classes. These classes are swigged to Python in the same manner as other Basilisk classes. This solves the interface issue because users can assign Python variables to the instantitions of these classes and access the attributes of those classes, include message history, directly. Additionally, pulling and plotting these logs is effectively instantaneous compared to pulling and plotting message logs in the original system.

- Disadvantage: Error proponesity and weak error handling due to string-type message identifies

  Solution: Because messages and related classes are accessed and connected as instantiations of Python classes, it is impossible to connect to messages that do not or will not exist due to a typo. In fact, this new system maintains the readability and simplicity of the of the original system because Python variables are also human-readable strings. Additionally, messages throw their own errors and can provide clear information about where that error came from.

- Disadvantage: Weak message typing

  Solution: The new, class-based, messaging system takes advantage of the C++ and Python type systems at both compile and run time. The main mechanism to take advantage of these features is C++ templating.

- Disadvantage: Implicit connections

  Solution: Because the connections made with this system are based on classes, those classes must be instantiated before a message can be connected. To be more clear, message connections are made by sharing pointers to message data. Therefore, there can be no default message connection because there is not memory to point to until a message has been instantiated.

- Disadvantage: Reference by message type

  Solution: As stated in the item above, the messages are connected by sharing pointers to message data. No lookup by message name, ID, or type is needed.

- Disadvantage: C types only

  Solution: Any data type that can be contained in a C++ struct is allowed in this system. If a message type is to be written or read by a C model, that message type should restric itself to variable types that are allowable in C.

- Disadvantage: Inherently Single-threaded

  Solution: Messages not only contain information in discrete locations, but also have their own methods for reading and writing. This does not enable multi-threading but, unlike the original singleton design pattern, it also does not prevent multi-threading.

DESIGN OF THE NEW SYSTEM

The new messaging system is based on templated C++ functors. A functor, also known as a function object, is a class with an overloaded parenthetical operator. For the sake of this work, a templated class is a class which only accepts a particular other class as method parameters and stores only that class.[6]

Message Class

The message class is the core of the new Basilisk messaging system. It is a templated class that encapsulates a message type data structure (the payload). Messages have three methods. These methods provide other new strongly typed classes that round out the system to read, write, and log a message. They are described below.

```
template<typename message_type>
class SimMessage{
public:
    std::string name = "unnamed message";  //!< -- can and should be
       assigned to aid in debugging
    SysProcess* process = nullptr;  //!< -- process that this message
       lives in
    message_type payload;  //!< -- message struct defining message
       payload
public:
    ReadFunctor<message_type> get_reader(){
        if (this->process){
            return ReadFunctor<message_type>(&this->payload, this->
                process);
        }else{
            std::cout << "Tried to get a reader from " << this->name <<
                ", but this message has not been set to a process.
                Quitting" << std::endl;
            std::exit(1);
        }
    }
    WriteFunctor<message_type> get_writer(){
        if (this->process){
            return WriteFunctor<message_type>(&this->payload, this->
                process);
        }else{
            std::cout << "Tried to get a writer from " << this->name <<
                ", but this message has not been set to a process.
                Quitting" << std::endl;
            std::exit(1);
        }
    }//!< Give write access
    Log<message_type> log(){return Log<message_type>(this);}
};
```

Read Functor

The read functor has a pointer to the message payload it can read. The overloaded parenthetical operator returns a copy of the message being read. The log method can also be used to make sure that a user is recording a history that reflects what a model was receiving as input. Generally, a read functor is a member of a Basilisk SysModel class used to provide inputs to the model. However, users can instantiate a reader via a message's `get_reader()` method to read a message directly from Python.

```
/*! Read functors have read-only access to messages*/
template<typename message_type>
class ReadFunctor{
private:
    message_type* payload_pointer = nullptr;
public:
    ReadFunctor() {}
    ReadFunctor(message_type* payload_pointer, SysProcess* proc) :
        payload_pointer(payload_pointer), process(proc){}
    bool linked(){return this->payload_pointer;}  //!< true if have
        an initialized payload pointer
    Log<message_type> log(){return Log<message_type>(this);}
    const message_type& operator()(){
        if (this->payload_pointer){
            return *this->payload_pointer;
        }else{
            std::cout << this->name << " tried to read but is
                uninitialized.";
            std::exit(1);
        }
    }
    void subscribeTo(SimMessage<message_type>* msg){
        if (this->process == msg->process){
            this->payload_pointer = &msg->payload;
        }else{
            std::cout << this->name << " tried to subscribe to " <<
               msg->name << "." << std::endl;
            std::cout << "However, " << this->name << " is in " <<
               this->process->name << " and " << msg->name << " is
               in " << msg->process->name << std::endl;
            std::cout << "Quitting." << std::endl;
            std::exit(1);
        }
    }
public:
    std::string name = "unnamed ReadFunctor";
    SysProcess* process = nullptr;
};
```

Write Functor

The write functor has a pointer to the message payload it can write to. The overloaded parenthetical operator takes a message type instance as a parameter. For the most part, write functors are contained in Basilisk SysModels and paired with messages. However, users can also instantiate write functors by calling `get_writer()` on a message in order to set a message from Python.

```
    template<typename message_type>
    class WriteFunctor{
    private:
        message_type* payload_pointer = nullptr;
    public:
        WriteFunctor(){}
        WriteFunctor(message_type* payload_pointer, SysProcess* proc) :
            payload_pointer(payload_pointer), process(proc){}
        void operator()(message_type payload){
            if (this->payload_pointer) {
                *this->payload_pointer = payload;
            }else{
                std::cout <<this->name << " tried to write but is
                   uninitialized.";
                std::exit(1);
            }
        }
        bool linked(){return this->payload_pointer;}
    public:
        std::string name = "unnamed write functor";
        SysProcess* process = nullptr;
    };
```

Log Class

The log class is a Basilisk SysModel class which records the history of a message throughout a simulation. A log can be constructed with either a read functor or message class as the constructor parameter. Alternatively, the `.log()` method can be called on either of those classes to return a log instance. A log, like all other SysModels must be added to a task which controls the rate at which the message will be recorded. Additionally, the log class records the times it was called, which is helpful for plotting the message history over time.

```
/*! A SysModel that keeps a time history of a message */
template<typename message_type>
class Log : public SysModel{
public:
    void initialize() override {};
    void Reset(uint64_t CurrentSimNanos) override {this->log.clear()
        ; this->log_times.clear();}  //!< only reset to 0
    void UpdateState(uint64_t CurrentSimNanos) override {
        this->log_times.push_back(CurrentSimNanos);
        this->log.push_back(this->read_message());
    }  //!< -- Read and record the message at the task rate of the
        task this log is in

    Log(){};
    ~Log(){};
    Log(SimMessage<message_type>* message){this->read_message =
        message->get_reader();}  //!< log a message
    Log(ReadFunctor<message_type>* message_reader){this->
        read_message = *message_reader;}  //!< log a reader
    std::vector<uint64_t>& times(){return this->log_times;}
    std::vector<message_type>& record(){return this->log;}

private:
    std::vector<message_type> log;
    std::vector<uint64_t> log_times;

private:
    ReadFunctor<message_type> read_message;
};
```

RESULTS

Code

The code a user writes as a result of this new system is significantly different than what was written before. The first code block below lists message interface lines for a SimpleNav model, `sNav`, in the original system while the second code block does this for the new system.

- Original System:

```
attErrorConfig.inputNavName = sNav.outputAttName
scSim.TotalSim.logThisMessage(sNav.outputTransName,
    samplingTime)
dataPos = scSim.pullMessageLogData(sNav.outputTransName + "
    .r_BN_N", list(range(3)))
```

- New System:

```
sNav.read_input_state.subscribeTo(sc.state_out_msg)  //
    this line is implicit in the original system
sNav.read_sun_input.subscribeTo(sunMsg)  // this line is
    implicit in the original system
snLog = self.sNav.output_trans_message.log()
task.AddNewObject(self.snLog)  // log timing is handled by
    tasks rather than a separate system
attError.readNavMsg.subscribeTo(sNav.output_att_message)
```

In the code blocks above, some users might prefer the lower line-count of he original system while some prefer the verbosity of the second. The other interesting thing to note is that there is no need to "pull" a message log in the new system because there is a variable assigned to that log data when the log is created.

Speed

Many of the features considered here to be advantages of the new system or disadvantages of the original system are hard to quantify. For instance, it is not clear how to apply a number to the error handling capability of one system over the other. The naturate of Basilisk makes error handling difficult generally due to the use of inheritance in storing models in tasks as well as the propagation of error handling through the Swig interface layer. Examination of the code and experience using it will lead different users to different conclusions in some cases, which leaves open the possibility for continued improvement.

However, the speed (run time) and efficiency (number of function calls) of the system can be measured quite straight-forwardly. This is displayed in table 1. In this table, both systems were used to run the script scenarioAttitudeFeedback.py, retrieve data via message logs, and plot the data. The plots were not displayed, though, so that the user interfacing with the plot would not affect the recorded speeds.

Table 1. Quantitative comparison of new and original messaging systems.

System	Run Time [s]	Primitive Calls
Original	0.912	561361
New	0.430	421897

The new system runs in less than half the time of the original system. If messages are not pulled in the original scenario, the run time drops to 0.6 seconds. This shows that accessing message logs is expensive in the original system, but the new system provides a run time speedup as well.

FUTURE WORK

There exists ample opportunity to build on and improve the work done here. Even if the new system is left as is, there are hours and days of work to be completed to fully implement this messaging system in the full Basilisk framework. Some opportunities for improvement also come to mind. First, error handling has been implemented in the code here in a less-than-optimal way in which the developer and/or user is required to name all messaging-related models in order that they provide useful information before crashing the simulation if used inappropriately. Second, the message payloads should be better-encapsulated. Third, a "bridge" class should be implemented which allows for the communication of these messages across SysProcess boundaries; as currently implemented, many spacecraft can be simulated in parallel, but without any knowledge of one another. Additionally, if knowledge of the task time step and simulation duration is provided, message logs could reserve memory rather than dynamically requesting it during the simulation. These are but a few of the improvements currently envisioned.

CONCLUSION

A new messaging system has been created for Basilisk. It aims to improve both the developer and user experience of the framework. Additionally, it improves the speed and makes Basilisk multi-threadable. It takes advantage of features of C++ to allow more useful message types and type safety. When it fails, it has the capability to provide verbose information about the failure, but there is room for error-handling improvement. Overall, the new system accomplished improvements over the original, which providing new value-added features to enrich the Basilisk framework.

REFERENCES

[1] T. Teil, H. Schaub, and S. Piggott, “Removing Rate Unobservability In Sun-Heading Filters Without Rate Gyros,” *AAS Spaceflight Mechanics Meeting*, Maui, Hawaii, January 13–17 2019, pp. 1563–1582. Paper No. AAS-19-460.

[2] T. Teil, H. Schaub, and S. Piggott, “Comparing Coarse Sun Sensor Based Sequential sun heading Filters,” *AAS Guidance and Control Conference*, Breckenridge, CO, Feb. 1–7 2018, pp. 395–406. Paper AAS 18-011.

[3] C. S. Lim and A. Jain, “Dshell++: A Component Based, Reusable Space System Simulation Framework,” *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, Pasadena, CA, USA, IEEE, July 2009, pp. 229–236, 10.1109/SMC-IT.2009.35.

[4] P. W. Kenneally, S. Piggott, and H. Schaub, “Basilsk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework,” DLR Oberpfaffenhofen, Germany, Nov. 2018.

[5] M. C. Margenet, P. Kenneally, H. Schaub, and S. Piggott, “Black Lion: A Software Simulator For Heterogeneous Spaceflight and Mission Components,” Breckenridge, CO, Feb. 2018.

[6] B. Stroustrup, *A Tour of C++*. C++ In-Depth Series, Pearson, 2 ed., 2018.