
HELICS Documentation

Philip Top, Trevor Hardy, Ryan Mast, Dheepak Krishnamurthy, And

Nov 25, 2021

CONTENTS

1	User Guide	3
1.1	Orientation	3
1.2	HELICS Installation	4
1.3	HELICS User Tutorial	6
1.4	Examples	73
1.5	Support	171
2	Developer Guide	173
2.1	Style Guide	173
2.2	Generating SWIG extension	175
2.3	Run tests	175
2.4	Generating Documentation	175
2.5	HELICS Benchmarks	176
2.6	Description of the different continuous integration test setups running on the CI servers	178
2.7	(Planned) CI/CD Infrastructure	180
2.8	Porting Guide: HELICS 2 to 3	183
2.9	Public API	186
2.10	RoadMap	188
2.11	HELICS Type Conversions	189
3	References	195
3.1	Tools with HELICS Support	195
3.2	Built-In HELICS Apps	197
3.3	Configuration Options Reference	219
3.4	API Reference	246
4	Quick links	247

[chat](#) [on gitter](#)

This is the documentation for the Hierarchical Engine for Large-scale Infrastructure Co-Simulation (HELICS). HELICS is an open-source cyber-physical-energy co-simulation framework for energy systems, with a strong tie to the electric power system. Although, HELICS was designed to support very-large-scale (100,000+ federates) co-simulations with off-the-shelf power-system, communication, market, and end-use tools; it has been built to provide a general-purpose, modular, highly-scalable co-simulation framework that runs cross-platform (Linux, Windows, and Mac OS X) and supports both event driven and time series simulation. It provides users a high-performance way for multiple individual simulation model “federates” from various domains to interact during execution—exchanging data as time advances—and create a larger co-simulation “federation” able to capture rich interactions. Written in modern C++ (C++14), HELICS provides a rich set of APIs for other languages including Python, C, Java, and MATLAB, and has native support within a growing number of energy simulation tools.

Brief History: HELICS began as the core software development of the Grid Modernization Laboratory Consortium (GMLC) project on integrated Transmission-Distribution-Communication simulation (TDC, GMLC project 1.4.15) supported by the U.S. Department of Energy’s Offices of Electricity Delivery and Energy Reliability (OE) and Energy Efficiency and Renewable Energy (EERE). As such, its first use cases center around modern electric power systems, though it can be used for co-simulation in other domains. HELICS’s layered, high-performance, co-simulation framework builds on the collective experience of multiple national labs.

Motivation: Energy systems and their associated information and communication technology systems are becoming increasingly intertwined. As a result, effectively designing, analyzing, and implementing modern energy systems increasingly relies on advanced modeling that simultaneously captures both the cyber and physical domains in combined simulations. It is designed to increase scalability and portability in modeling advanced features of highly integrated power system and cyber-physical energy systems.

USER GUIDE

Co-simulation is a powerful analysis technique that allows simulators of different domains to interact through the course of the simulation, typically by dynamically exchanging information that defines boundary conditions for other simulators. HELICS is a co-simulation platform that has been designed to allow integration of these simulators across a variety of computation platforms and languages. HELICS has been designed with power system simulation in mind (GridLAB-D, GridDyn, MATPOWER, OpenDSS, PSLF, InterPSS, FESTIV) but is general enough to support a wide variety of simulators and co-simulation tasks. Support for other domains is anticipated to increase over time.

1.1 Orientation

There are a number of classes of HELICS users:

- **New users** that have little to no experience with HELICS and co-simulation in general
 - Start with *Installation*
 - Read the *Fundamental Topics*
 - Try the *Examples*
- **Intermediate users (Modelers)** that have run co-simulations with HELICS using simulators in which somebody else has done the simulator integration with HELICS.
 - Review *Fundamental Topics* as needed
 - Look over the *Advanced Topics* to see which features of HELICS may be most useful for your analysis.
 - * *Multi-Source Inputs (example)*
 - * *Broker Hierarchies (example)*
 - * *HELICS Core Types (example)*
 - * *Queries (example)*
 - * *Simultaneous Co-simulation (example)*
 - * *Multiple Co-simulation Orchestration (example)*
- **Experienced users (Integrators)** that are incorporating a new simulator and need to know how to use specific features in the HELICS API
 - Look in the *Configurations Options Reference* or jump straight to the API references
 - * C++
 - * C++98
 - * C

- * [Python](#)
- * [Julia](#)
- * [nim](#)

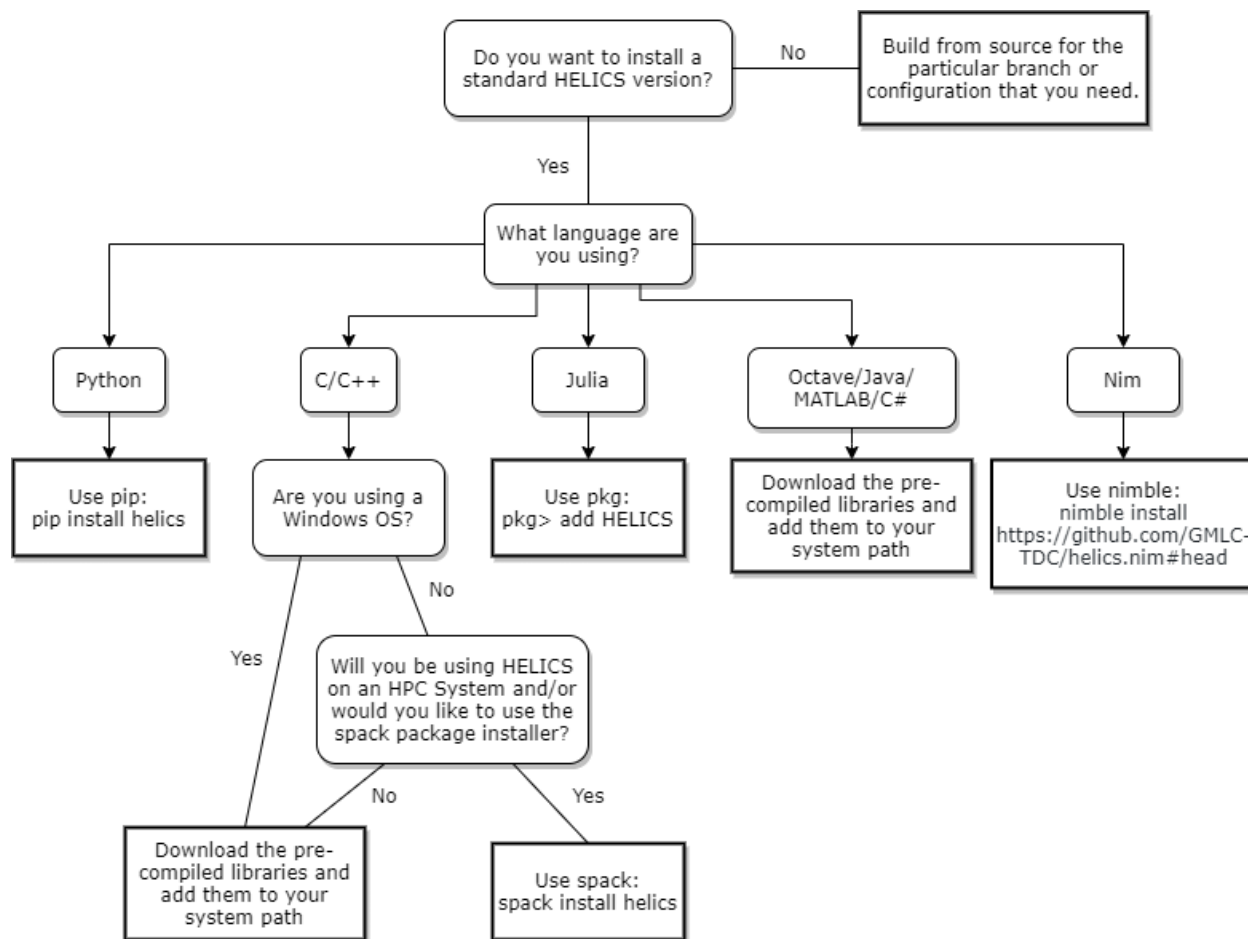
- **Developers** of HELICS who are improving HELICS functionality and contributing to the code base
 - See the *[Developer Guide](#)*

1.2 HELICS Installation

1.2.1 HELICS Installation Methods

The first step to using HELICS is to install it. Since there are several ways to do this, the flow chart below provides some insight into what approach is likely to be the easiest depending upon a number of factors, most predominantly the programming language bindings that you intend to use. Below the flow chart you'll provide links to more complete instructions for each method. Note that you'll need an internet connection for this process, as we'll be downloading HELICS from the internet.

As of HELICS v3, the only supported language bindings that are included with the core HELICS library downloads are C and C++98, in addition to C++17 when building from source. If you end up needing to build from source AND use one of the supported language bindings you'll need to follow the instructions for installing HELICS for said language. This would also be the case if you were needing to run a co-simulation that used tools that provided their HELICS implementation in a variety of languages. Generally speaking, as long as all supported languages are on similar versions, each one can use its own installed version of HELICS without any trouble. The supported languages also have ways of being pointed towards a specific HELICS installation (rather than the one they install for themselves) if that is preferred or necessary for a particular use case.



pip install

`pip install helics`

Download pre-compiled

Download the pre-compiled libraries and add them to your system path

spack install

spack install helics

nimble install

nimble install <https://github.com/GMLC-TDC/helics.nim#head>

Build from source

Build from source

1.2.2 helics-cli Installation

`helics-cli` is a supporting tool that provides a simple and standardized way of launching HELICS-based co-simulations. This tool is used for *all the examples in this User Guide* and thus its installation is highly recommended. (`helics-cli` also provides access to the *web interface for monitoring and debugging co-simulations*, another good reason to install it.)

Installation of `helics-cli` is straightforward:

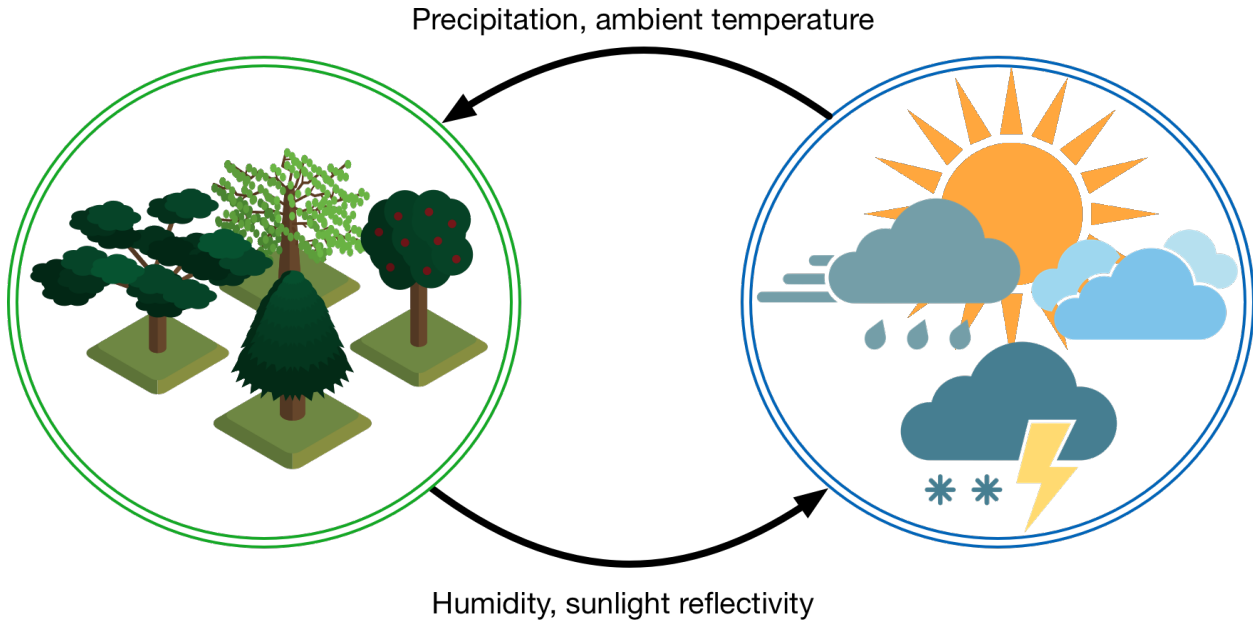
```
pip install git+git://github.com/GMLC-TDC/helics-cli.git@main
```

1.3 HELICS User Tutorial

1.3.1 Co-Simulation Overview

Co-simulation is an analysis technique that allows simulators from different domains to interact with each other, typically by exchanging values through the course of the simulation that define other simulators' boundary conditions. For example, you may have a model and simulator that is able to describe atmospheric conditions and weather as they progress through time. You may have just found out about another well-established model and simulator that describes the growth of vegetation over large geographic areas. In reality, these two systems clearly interact with the precipitation and temperatures impacting the growth of vegetation and the amount of vegetation impacting air temperature and moisture content. These models, though, may treat this interaction as a boundary condition such that the atmospheric simulator assumes a fixed rate of sunlight reflection from the ground and the vegetation simulator uses a time-series of historical precipitation values.

Co-simulation allows the coupling of existing models and simulators by breaking down these boundary condition barriers and calculating results that are more realistic and dynamic. In this case, while the atmospheric simulator is able to calculate the current weather using the latest values of surface reflectivity and humidity contributions from the vegetation, the vegetation simulator is able to use the latest values from the atmospheric simulator to determine the moisture content of the soil and the ambient temperature.



It's easy to imagine this hypothetical model being extended with other simulators throughout the ecosphere, incorporating models describing the behavior of the ocean, wildlife, even human agriculture. Rather than trying to build a single simulator that models all of this functionality, co-simulation allows us to use existing simulators that have longer-term investment and validation behind them; that experience is leveraged into building simulations of complex systems of systems.

Co-simulation also has a built-in parallelism that allows it to be used to scale up to very large models. Particularly for systems with limited interactions between portions of the models, the value exchanges that take place during co-simulation are limited, allowing the individual instances of the simulators to run independently. Using our previous example, it is easy to imagine that there is limited to no interaction between vegetation in Oregon and vegetation in Texas (except through the atmosphere). Simulations of the vegetations in each of these states can run independently due to this lack of interaction, each providing values to the atmospheric model that indirectly couples them together. When able to be constructed in such a manner, the limitations on the size and complexity of the co-simulation become limited more by computing resources than by the capabilities of the individual simulators.

This being said, coupling these simulators together effectively can be summarized in the execution of two very related functions:

1. Maintaining synchronization of all the simulator instances running
2. Facilitating the data transfer between them

Maintaining synchronization is required when working in a heterogeneous simulation environment where each simulator's concept of time is different and the computation time required by each simulator instance can vary widely. Without the regulation of the universal co-simulation time, individual simulator instances could easily run ahead of the rest, simulating days and weeks ahead of the others.

And when one simulator instance is simulating one week ahead of the others, it becomes impossible for the results from that simulation to affect the rest of the simulator instances; the values it passes and receive are literally from a different point in time than the rest and are effectively meaningless. Without the synchronization of time, there is no way for the simulator instances to affect each other and without this interaction, the results of the co-simulation are meaningless.

HELICS, at its core, has been designed to do these two functions as efficiently, quickly, and comprehensively as possible to support a wide variety of simulators and models. Simulators with HELICS support have been modified in such a way so as to allow HELICS (really, a HELICS core object but we'll get to that in the [section on timing](#)) to advance the time of the simulator and provide it new values for specific variables that it is interested in. Making this modification

to the simulator is not necessarily difficult but it must be done in such a way as to allow for these two fundamental functions to be executed.

Even after a simulator has been modified to support HELICS, users of that simulator who are building co-simulations have the task of constructing or designing a co-simulation to perform the particular analysis they desire. Many simulators have thousands of potential values they can provide to other simulators and generally, most of these values will not be needed by other simulators. It is the role of those designing the co-simulation to correctly configure each of the simulator instances such that the appropriate values are sent and received by the simulator instances and are mapped to the appropriate parameters inside each simulator. For a small number of simulators with a small number of values, this is not necessarily difficult but as the size of the co-simulation grows, this task becomes more challenging.

Thinking about HELICS co-simulation in general and about the nature of the simulators that have been or need to be integrated with HELICS, there are a few questions to consider. Though these questions have most immediate impact as you being planning on how to integrate a particular simulator they are also useful when thinking about how simulators in a federation need to be made to interact properly.

1. **What is the nature of the code-base being integrated?** Is this open-source code that can be fully modified? Is it a simulator, perhaps commercial, that provides an API that will be used? How much control do you, the integrator, have in modifying the behavior of the simulator? A number of existing simulation tools have *already been integrated into HELICS* and can serve as good references when considering how an integration might be accomplished.
2. **What programming language(s) will be used?** - HELICS has bindings for a number of languages and the one that is best to use may or may not be obvious. If your integration of the simulator will be through the API of the existing simulator, then you'll likely be writing a standalone executable that wraps that API. You may be constrained on the choice of languages based on the method of interaction with that API. If the API is accessed through a network socket then you likely have a lot of freedom in language choice. If the API is a library that you call from within wrapper, you will likely be best served using the language of that library.

If you're writing your own simulator then you have a lot more freedom and the language you use may come down to personal preference and/or performance requirements of the federate.

The languages currently supported by HELICS are:

- C++
 - C
 - Python (2 and 3)
 - Java
 - MATLAB
 - Octave
 - C# (somewhat limited)
 - Julia
 - Nim
3. **What is the simulator's concept of time?** - Understanding how the simulator natively moves through time is essential when determining how time requests will need to be made. Does the simulator have a fixed time-step? Is it user-definable? Does the simulator have any concept of time or is it event-based?
 4. **What is the nature of the data the simulator will send to and receive from the rest of the federation?** Often, this answer is in large part provided by the analysis need that is motivating the integration. However, there may be other angles to consider beyond what's immediately apparent. As a stand-alone simulator, what are its inputs and outputs? What are its assumed or provided boundary conditions? Where do interdependencies exist between the simulator and other simulators within the federation? What kinds of data will it be providing to the rest of the federation?

1.3.2 Fundamental Topics

HELICS Terminology

Before digging into the specifics of how a HELICS co-simulation runs, there are a number of key terms and concepts that need to be clarified first.

- **Simulator** - A simulator is the executable that is able to perform some analysis function, often but not always, by solving specific problems to generate a time series of values. Simulators are abstract in the sense that it largely refers to the software in a non-executing state, outside of the co-simulation. We might say things like, “That simulator doesn’t model xyz appropriately for this analysis.” or “This simulator has been parallelized and runs incredibly quickly on many-core computers.” Any time we are talking about a specific instance of a simulator running a specific model you are really talking about a...
- **Federate** - Federates are the running instances of simulators that have been assigned specific models and/or have specific values they are providing to and receiving from other federates. For example, we can have ten distribution circuit models that we are connecting for a co-simulation. Each could be run by the simulator GridLAB-D, and when they are running, they become ten unique federates providing unique values to each other. A collection of federates working together in a co-simulation is called a “federation.”
- **Model** - A model is synonymous with an agent in the co-simulation. A simulator contains the calculations for the model. Depending on the needs of the co-simulation, a federate can be configured to contain one or many models. For example, if we want to create a co-simulation of electric vehicles, we may write a simulator (executable program) to model the physics of the electric vehicle battery. We can then designate any number of agents/models of this battery by configuring the transfer of signals between the “Battery Federate” (which has N batteries modeled) and another federate.
- **Signals** - Signals are the the information passed between federates during the execution of the co-simulation. Fundamentally, co-simulation is about message-passing via these signals. HELICS divides these messages into two types: value signals and message signal. The former is used when coupling two federates that share physics (e.g. batteries providing power to wheel motors on an electric car) and the later is used to couple two federates with information (e.g. a battery charge controller and a charge relay on a battery). There are various techniques and implementations of the message-passing infrastructure that have been implemented in the core. There are also a variety of mechanisms within a co-simulation to define the nature of the data being exchanged (data type, for example) and how the data is distributed around the federation.
- **Interface** - A structure by which a federate can communicate with other federates. Includes Endpoints, Publications, and Inputs.
- **Core** - The core is the software that has been embedded inside a simulator to allow it to join a HELICS federation. In general, each federate has a single core, making the two synonymous (core \Leftrightarrow federate). The two most common configurations are: (1) one core, one federate, one model; (2) one core, one federate, multiple models. There are sometimes cases where a single executable is used to represent multiple federates and all of those federates use a single core (one core, multiple federates, multiple models). Cores are built around specific message buses with HELICS supporting a number of different bus types. Selection of the message bus is part of the configuration process required to form the federation. Additional information about cores can be found in the [Advanced Topics](#).
- **Broker** - The broker is a special executable distributed with HELICS; it is responsible for performing the two key tasks of a co-simulation: (1) maintaining synchronization in the federation and (2) facilitating message exchange. Each core (federate) must connect to a broker to be part of the federation. Brokers receive and distribute messages from any federates that are connected to it, routing them to the appropriate location. HELICS also supports a hierarchy of brokers, allowing brokers to pass messages between each other to connect federates associated with different brokers and thus maintain the integrity of the federation. The broker at the top of the hierarchy is called the “root broker” and it is the message router of last resort.

Federates

This section on Federates covers:

- *What is a Federate?*
- *Types of Federates*
 - *Value Federates*
 - * *Value Information*
 - * *Value Federate Interfaces*
 - *Message Federates*
 - * *Message Information*
 - * *Message Federate Interfaces*
 - *Endpoints*
 - *Native HELICS Filters*
 - * *Interactions Between Messages and Values*
 - *Combination Federates*

What is a Federate?

A “federate” is an instance of a simulation executable that models a group of objects or an individual objects. For example, one can write a simulator to model the battery of an electric vehicle (EV). If we want to model multiple EVs connected to charging ports in a dedicated EV charging garage, we can use the EV simulator to model a group of EVs. We will need a second simulator to model a group of charging ports. Once we launch the simulators, each is called a “federate”. Together, they are called a “federation” – multiple federates running simultaneously. This federation performs a co-simulation to achieve a particular analysis objective (*e.g.* replicate the behavior of a fleet of EVs charging to understand the charging power requirements).

This co-simulation is described in more detail in the *Fundamental Examples*. There are two federates in this co-simulation; one modeling the batteries on board the EVs, and one modeling the chargers of the batteries. Each federate in this example has multiple objects it is modeling; five batteries for the battery federate, and five chargers for the charger federate. Because the objects being modeled with the federates share most of the same properties – *e.g.* battery size, charge rate – a single federate can be used to model the five batteries. As the complexity of the co-simulation increases, it becomes increasingly difficult to group objects into one federate. In these situations, we could also design the co-simulation with one federate for each EV.

Co-simulations are designed to answer a research question. The question addressed by the *Fundamental Examples* is: How much power is needed to serve five EVs in a dedicated charging garage?

With this research question, we have identified that we want to model **batteries** and **chargers** and we want to monitor the power draw in **kW** over time. It’s important to first identify the types of objects you want to model, as co-simulation in HELICS requires constructing federates based on the type of information they pass to other federates.

Types of Federates

Federates are distinguished by the types of information they model and the interfaces they use to pass the information. Interfaces define how signals are connected between federates in a federation. HELICS has three types of federates: **Value Federates**, **Message Federates**, and **Combination Federates**. Value federates model physics in a system, message federates model logic (i.e., controls), and combination federates model both.

Value Federates

Value federates are used when the federate is simulating the physics of a system. The data in the messages they send and receive indicate new values at the boundary of the federate system under simulation. Value federates interface with the federation using one-to-one correspondence to internal variables within the federate. These interfaces are commonly called publications and subscriptions (pubs/subs).

Value Information

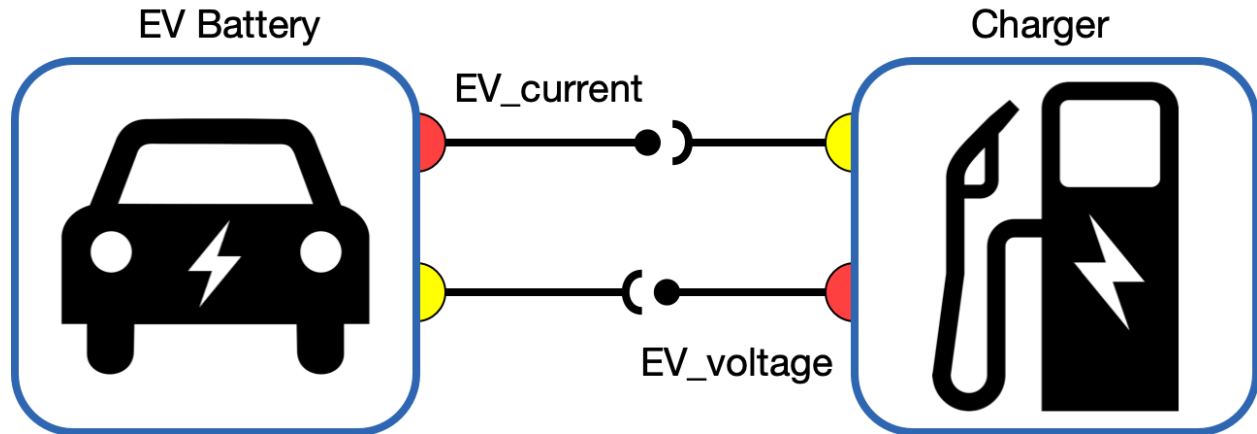
The information modeled by value federates is physical values in a system with associated units. In the *Fundamental Example*, the batteries and chargers are both value federates; charger applies a voltage to the battery (Charger federate sending the Battery federate that value) and the Battery federate responds with the charging current which it sends back to the Charger. These two federates each update the other's boundary condition's state: the voltage and current. Federates that model physics must be configured as **value federates**. Value federates typically will update at very regular intervals based on the fidelity of their models and/or the resolution of any supporting data they are using.

Value Federate Interfaces

Value federates have direct fixed connections through interfaces to other federates. There are three interface types for value federates that allow the interactions between the federates to be flexibly defined. The difference between the three types revolves around whether the interface is for sending or receiving values and whether the sender/receiver is defined by the federate:

- Publications
 - Sending interface
 - Handle named with "key" in configuration
 - Recipient handle of value is not necessary, however it can be specified with "targets" in configuration
- Subscriptions (Unnamed Inputs)
 - Receiving interface
 - Not "named" (no identifier to the rest of the federation)
 - Source of value handle is specified with "key" in configuration
- Named Inputs
 - Receiving interface
 - Handle named with "key" in configuration
 - Source handle of value is not necessary, however it can be specified with "targets" – Named Inputs can receive values from multiple "targets"

The most commonly used of these fixed interfaces are publications and subscriptions. In the *Fundamental Example*, the Battery federate and the Charger federate have fixed pub/sub connections. In the figure below, publishing interfaces are in **red** and the subscription interfaces are in **yellow**. The Battery federate **publishes** the current flowing into the battery from the publication interface named EV_Battery/EV_current and does not specify the intended recipient. The Charger federate **subscribes** to the amps from the Battery with the subscription interface named EV_Battery/EV_current – the receiving interface only specifies the sender.



In all cases the configuration of the federate core declares the existence of the interface to use for communicating with other federates. The difference between publication/subscription and directed outputs/unnamed inputs is where the federate core knows the specific names of the interfaces on the receiving/sending federate core.

The interface type used for a federation configuration is a preference of the user setting up the federation. There are a few important differences that may guide which interfaces to use:

- Which interfaces does the simulator support?
 - Though it is the preference of the HELICS development team that all integrated simulators support all types, that may not be the case or even possible. Limitations of the simulator may limit your options as a user of that simulator.
- Is portability of the federate and its configuration important?
 - Because publications and subscriptions (unnamed inputs) don't require the federate to know who it is sending HELICS messages to and receiving HELICS messages from, it affords a slightly higher degree of portability between different federations. The mapping of HELICS messages still needs to be done to configure a federation, it's just done separately from the federate configuration file via a broker or core configuration file. The difference in location of this mapping may offer some configuration efficiencies in some circumstances.

Message Federates

Message federates send packets of data with unfixed connections for things such as events, communication packets, or triggers. Message federates are used to model information transfers (versus physical values) moving between federates. Measurement and control signals are typical applications for these types of federates.

Message Information

Message federates are used when the HELICS signals being passed to and from the simulation are generic packets of information, often for control purposes. They are treated as data to be used by an algorithm, processor, or controller. If the inputs to the federate can be thought of as traveling over a communication network of some kind, it should be modeled as coming from/going to a message federate. For example, in the power system world, phasor measurement units (PMUs) have been installed throughout the power system and the measurements they make are collected through a communication system and would be best modeled through the use of HELICS messages.

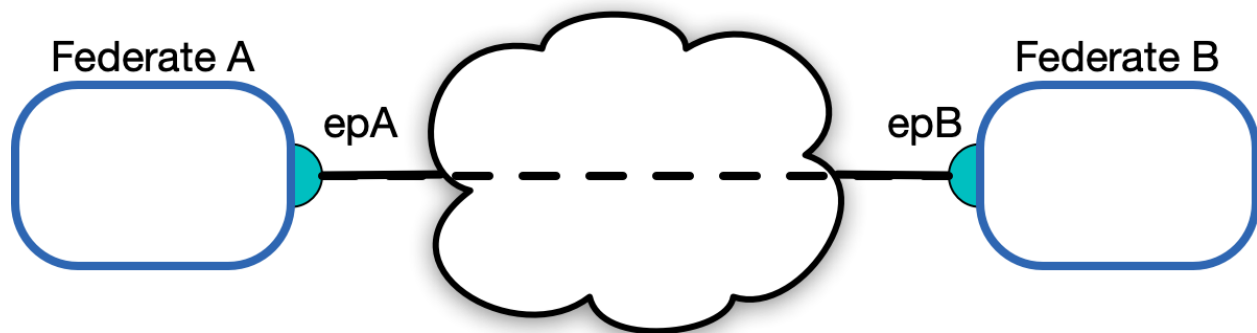
Message Federate Interfaces

Message federates interact with the federation through endpoints interfaces. Message federates can be thought of as attaching to communication networks, where the federate's **endpoints** are the specific interfaces to that communication network. By default, HELICS acts as the communication network, transferring messages between endpoint interfaces configured for message federates. Just as communications networks can be susceptible to failure, messages can be altered or delayed with **filter** which can be associated with specific endpoints. Filters can only act on messages and thus can only be associated with endpoints. Value signals are meant to replicate physical connections between models and thus are not susceptible to the frailty of communication systems.

Endpoints

Endpoints are interfaces used to pass packetized data blocks (messages) to another federate. Message federates interact with the federation by defining an endpoint that acts as their address to send and receive messages. Message federates are typically sending and receiving measurements, control signals, commands, and other signal data with HELICS acting as a perfect communication system (infinite bandwidth, no latency, guaranteed delivery).

In the figure below, Federate A and B are message federates with endpoints epA and epB. They do not have a fixed communication pathway; they have unique addresses (endpoints) to which messages can be sent. An endpoint can send data to any other endpoint in the system – it just needs the “address” (endpoint handle).



Endpoints can have a "type" which is a user defined string. HELICS currently does not recognize any predefined types. The data consists of raw binary data and optionally a send time. Messages are delivered first by time order, then federate id number, then handle id, then by order of arrival.

Unlike HELICS values which are persistent (meaning they are continuously available throughout the co-simulation), HELICS messages are only readable once when collected from an endpoint. Once that collection is made, the message only exists within the memory of the collecting message federate. If another message federate needs the information, a new message must be created and sent to the appropriate endpoint.

Native HELICS Filters

Filters are objects that can be used to disrupt message packets in a manner similar to communications networks. Filters are associated with the HELICS core, which in turn manages a federate's endpoints. Typical filtering actions might be delaying the transmission of a message or randomly dropping a certain percentage of the received messages. Filters can also be defined to operate on messages being sent ("source filters") and/or messages being received ("destination filters").

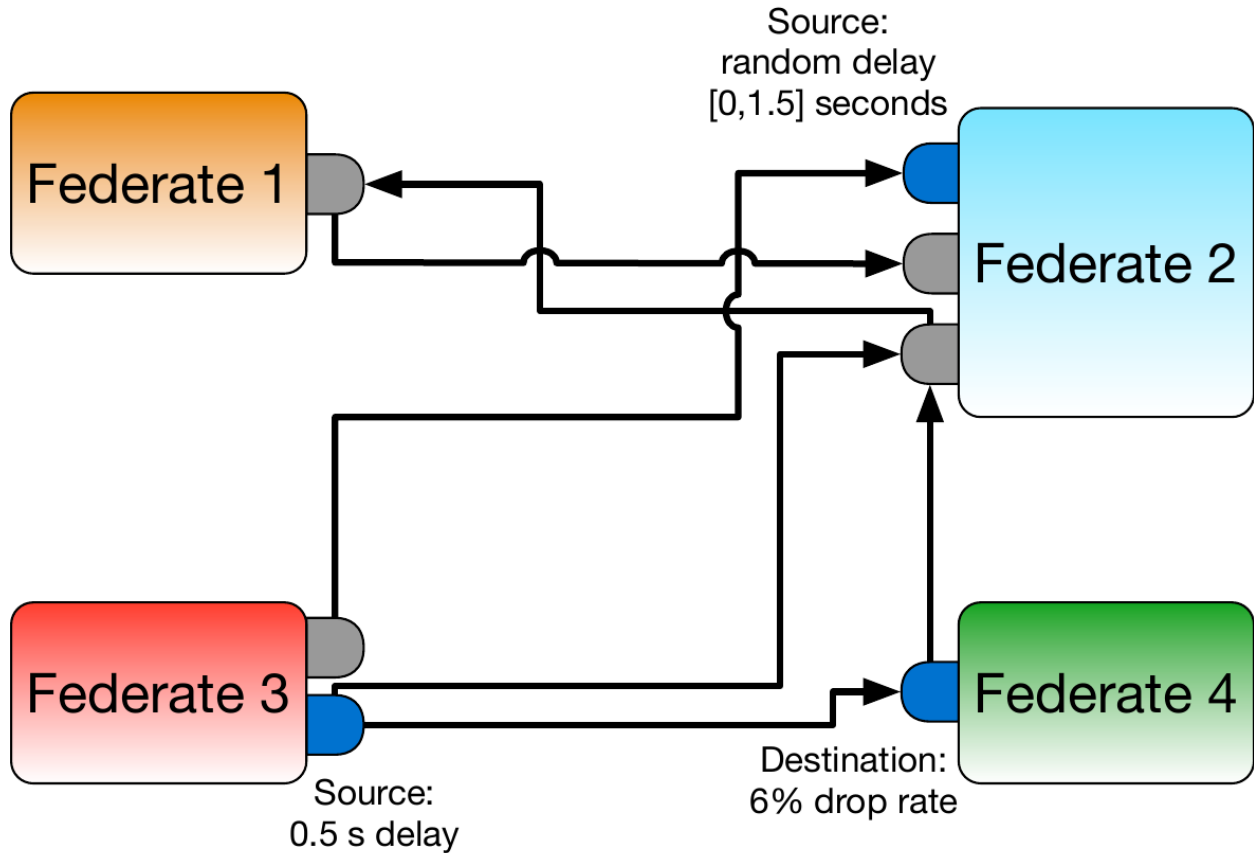
Messages can be filtered, values cannot. Messages are directed and unique, values are persistent. Internal to HELICS, each message has a unique identifier and can be thought to travel through a generic communication system between the source and destination endpoints. Since HELICS values model direct physical connections, they do not pass through this generic network and they cannot be operated on by filters. It is possible to create a federation where HELICS value interfaces are used to send control signals but this removes the possibility of using filters and the easy integration of communication system simulators.

Filters have the following properties:

1. Inline operations that can alter a message or events
 - Time Delay (Random or Fixed)
 - Packet Translation
 - Random Dropping
 - Message Cloning / Replication
 - Rerouting
 - Firewall
 - Custom
2. Filters are part of the HELICS core and the effect of a filter is not limited to the endpoints of local objects
3. A single Filter can be configured once and then applied to multiple endpoints as unique instances of that configuration. Filters can be triggered by either messages sent from an endpoint (source target) or messages received by an endpoint (destination targets)
4. Filters can be cloning or non-cloning filters. Cloning filters will operate on a copy of the message and in the simple form just deliver a copy to the specified destination locations. The original message gets delivered as it would have without the filter.

The figure below is an example of a representation of the message topology of a generic co-simulation federation composed entirely of message federates. Source and destination filters have been implemented (indicated by the blue endpoints – gray endpoints do not have filters), each showing a different built-in HELICS filter function.

Message Topology



- In this figure, Federate 4 has a single endpoint for sending and receiving messages. Both a source filter and a destination filter can be set up on a single endpoint, or multiple source filters can be used on the same endpoint.
- The source filter on Federate 3 delays the messages to both Federate 2 and Federate 4 by 0.5 seconds. Without establishing a separate destination endpoint devoted to each federate, there is no way to produce different delays in the messages sent along these two paths.
- Because the filter on Federate 4 is a destination filter, the message it receives from Federate 3 is affected by the filter but the message it sends to Federate 2 is not affected.
- The source filter on Federate 2 has no impact on this co-simulation as there are no messages sent from that endpoint.
- Individual filters can be targeted to act on multiple endpoints and act as both source and destination filters.

Interactions Between Messages and Values

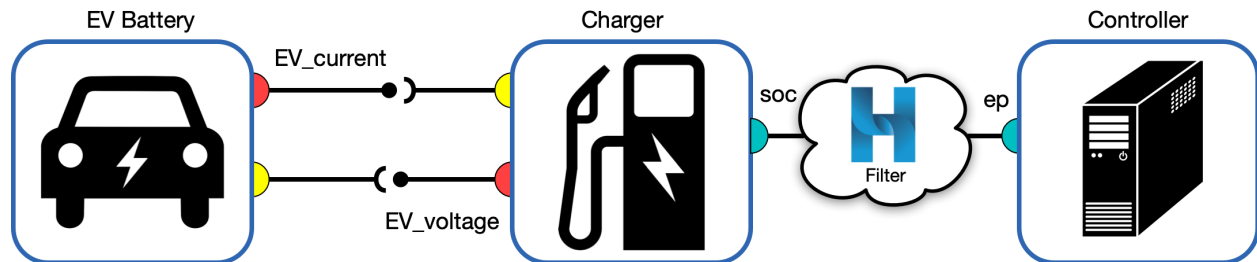
Because HELICS values are used to represent physical reality, they are available to any subscribing federate at any time. If the publishing federate only updates the value, say, once every minute, any subscribing federates that are granted a time during that minute window will all receive the same value.

Though it is not possible to have a HELICS message show up at a value interface, the converse is possible; message federates can subscribe to HELICS values. Every time a value federate publishes a new value to the federation, if a message federate has configured an endpoint with a "subscription" parameter defined, HELICS will generate a new HELICS message and send it directly to the subscribing endpoint every time a new value is published. These messages are queued and not overwritten (unlike in HELICS values) which means when a message federate is granted a time, it may have multiple messages from the same source to process.

This feature offers the convenience of allowing a message federate to receive messages from pure value federates that have no endpoints defined. This is particularly useful for simulators that do not support endpoints but are required to provide measurement signals for controllers. Implemented in this way, though, it is not possible to later implement a full-blown communication simulator that these values-turned-messages can traverse. Such co-simulation architectures in HELICS require the existence of both a sending and receiving endpoint; this feature very explicitly by-passes the need for a sending endpoint.

Combination Federates

Combination federates make use of both value signals and message signals for transferring data between federates. The *Combination Federation* in the Fundamental Examples learning track introduces a third federate to the *Base Example* – the combination federate passes values with the battery federate to monitor the physics of the battery charging, and it also passes messages with a controller federate to decide when to stop charging.



The following table may be useful in understanding the differences between the two methods by which federates can communicate:

	Values	Messages
Interface Type:	Publication/Subscription/Input	Endpoint/Filter
<i>Signal Route:</i>	Fixed, defined at initialization	Determined at time of transmission
Outgoing signal:	1 to n (broadcast)	1 to 1 - defined sender and receiver
Incoming signal:	n to 1 (promiscuous)	None
Status on Message Bus:	Current value always available	Removed when received
Fidelity:	Default value	Rerouting/modification through filters
Signal Contents:	Physical units	Generic binary blobs

Federate Interface Configuration

As soon as one particular instance of a simulator begins running in a co-simulation it is a federate. Every federate will require configuration of the way it will communicate (send signals) to other federates in the federation. For *simulators that already have HELICS support*, the configuration takes the form of a JSON (or TOML) file; bespoke simulators can be configured with the HELICS APIs in the code or via a JSON file. The essential information required to configure federate interfaces with HELICS is:

Federate name - The unique name this federate will be known as throughout the federation. It is essential this name is unique so that HELICS messages can route properly.

Core type - The core manages interfaces between the federation and the federate; there are several messaging technologies supported by HELICS.

Publications and Inputs - Publication configuration contains a listing of source handle, data types, and units being sent by the federate; input configuration does the same for values being received by the federate. If supported by the simulator (e.g., *a Python simulator*), these values can be mapped to internal variables of the simulator from the configuration file.

Endpoints - Endpoints are sending and receiving points for HELICS messages to and from message federates. They are declared and defined for each federate.

Time step size - This value defines the resolution of the simulator to prevent HELICS from telling the simulator to step to a time of which it has no concept (e.g. trying to simulate the time of 1.5 seconds when the simulator has a resolution of one second).

This section describes how to configure the federate interfaces using JSON files and API calls. Extensive details on the options for configuring HELICS federates is available in the *Configurations Options Reference*. If the user has written the simulator, it may be preferable to use the HELICS APIs to configure the federates, because the interface registrations can be made into a variable of the simulation. For non-open-source simulators, the JSON configuration files must be written before the federation is launched.

- *JSON Configuration*
- *API Configuration*

JSON Configuration

Federate interfaces must be configured with JSON files if they are built from non-open-source tools, such as the simulators listed in the reference page on *Tools with HELICS Support*. Federates built from simulators written by the user can be configured with JSON files or API calls.

The *Examples* illustrate in detail how to integrate federates built from open source tools, such as Python. The *Fundamental Combination Federation* configures the Python federate interfaces with JSON files. In this co-simulation, there are three federates: a value Battery federate, a combination Charger federate, and a message Controller federate. The example JSON shows the interface configuration for the combination federate to illustrate the different types of interfaces.

Sample JSON configuration file

The most common parameters are set in the file `ChargerConfig.json`. There are many, many more configuration parameters that this file could include; a relatively comprehensive list along with explanations of the functionality they provide can be found in the [Configurations Options Reference](#).

```
{
  "name": "Charger",
  "loglevel": 7,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": [
    {
      "name": "Charger/EV1.soc",
      "destination": "Controller/ep",
      "global": true
    },
    {
      ...
    }
  ],
  "publications": [
    {
      "key": "Charger/EV1_voltage",
      "type": "double",
      "unit": "V",
      "global": true
      "tags": {
        "period": 0.5,
        "description": "a test publication"
      }
    },
    {
      ...
    }
  ],
  "subscriptions": [
    {
      "key": "Battery/EV1_current",
      "type": "double",
      "unit": "A",
      "global": true
    },
    {
      ...
    }
  ]
}
```

JSON configuration file explanation

- **name** - Every federate must have a unique name across the entire federation; this is functionally the address of the federate and is used to determine where HELICS messages are sent. An error will be generated if the federate name is not unique.
- **loglevel** - The level of detail exported to the *log files* during run time ranges from 1 (minimal) to 7 (most).
- **coreType** - There are a number of technologies or message buses that can be used to send HELICS messages among federates, detailed in *Core Types*. Every HELICS enabled simulator has code in it that creates a core which connects to a HELICS broker using one of these messaging technologies. ZeroMQ (zmq) is the default core type and most commonly used, but there are also cores that use TCP and UDP networking protocols directly (forgoing ZMQ's guarantee of delivery and reconnection functions), IPC (uses Boost's interprocess communication for fast in-memory message-passing but only works if all federates are running on the same physical computer), and MPI (for use on high-performance computing clusters where MPI is installed).
- **period** - The federate needs instruction for how to step forward in time in order to synchronize calculations. This is the simplest way to synchronize simulators to the same time step; this forces the federate to time step intervals of $n \times \text{period}$. The default units are in seconds. Timing configuration is explained in greater detail in the *Timing* page, with additional configuration options in the *Configuration Options Reference*.
- **uninterruptible** - Setting `uninterruptible` to `false` allows the federate to be interrupted if there is a signal available for it to receive. This is a *timing configuration* option.
- **terminate_on_error** - By default, HELICS will not terminate execution of every participating federate if an error occurs in one. However, in most cases, if such an error occurs, the cosimulation is no longer valid. Setting `terminate_on_error` frees the federate from the broker if there is an error in execution, which simplifies debugging. This will prevent your federate from hanging in the event that another federate fails.
- **endpoints**
 - **name** - The string in this field is the unique identifier/handle for the endpoint interface.
 - **destination** - This option can be used to set a default destination for the messages sent from this endpoint. The default destination is allowed to be rerouted or changed during run time.
 - **global** - Just as in value federates, `global` allows for the identifier of the endpoint to be declared unique for the entire federation.
- **publications**
 - **key** - The string in this field is the unique identifier/handle (at the federate level) for the value that will be published to the federation. In the example above, `global` is set to `true`, meaning the `key` must be unique to the entire federation.
 - **global** - Indicates that the value in `key` will be used as a global name when other federates are subscribing to the message. This requires that the user ensure that the name is used only once across all federates. Setting `global` to `true` is handy for federations with a small number of federates and a small number of message exchanges as it allows the `key` string to be short and simple. For larger federations, it is likely to be easier to set the flag to `false`.
 - **required** - At least one federate must subscribe to the publications.
 - **type** - Data type, such as integer, double, complex.
 - **units** - The units can be any sort of unit string, a wide assortment is supported and can be compound units such as m/s^2 and the conversion will convert as long as things are convertible. The unit match is also checked for other types and an error if mismatching units are detected. A warning is also generated if the units are not understood and not matching. The unit checking and conversion is only active if both the publication and subscription specify units. HELICS is able to do some levels of unit conversion, currently only on double type publications but more may be added in the future.

- * **tags** - Arbitrary string value pairs that can be applied to interfaces. Tags are available to others through queries but are not transmitted by default. They can be used to store additional information about an interface that might be useful to applications. At some point in the future automated connection routines will make use of them. “tags” are applicable to any interface and can also be used on federates.
- **subscriptions** - These are lists of the values being sent to and from the given federate.
 - **key** - This string identifies the federation-unique value that this federate wishes to receive. If `global` has been set to `false` in the `publications` JSON configuration file, the name of the value is formatted as `<federate name>/<publication key>`. Both of these strings can be found in the publishing federate’s JSON configuration file as the `name` and `key` strings, respectively. If `global` is `true` the string is the publishing federate’s `key` value.
 - **required** - The message being subscribed to must be provided by some other publisher in the federation.
 - **type** - Data type, such as integer, double, complex.
 - **units** - Same as with publications.
 - **global** - Applies to the key, same as with publications.

API Configuration

Configuring the federate interface with the API is done internal to a user-written simulator. The specific API used will depend on the language the simulator is written in. Native APIs for HELICS are available in C++ and C. MATLAB, Java, Julia, Nim, and Python all support the C API calls (ex: `helicsFederateEnterExecutionMode()`). Python and Julia also have native APIs (see: [Python \(PyHELICS\)](#), [Julia](#)) that wrap the C APIs to better support the conventions of their languages. The [API References](#) page contains links to the APIs.

The [Examples](#) in this User Guide are written in Python – the following federate interface configuration guidance will use the [PyHELICS](#) API, but can easily be adapted to other C-based HELICS APIs.

Sample PyHELICS API configuration

The following example of a federate interface configuration with the PyHELICS API comes from the Fundamental Final Example. This co-simulation has exactly the same interface configuration as the Combination Federation above. The only difference is that the federate interfaces are configured with the PyHELICS API.

In the `Charger.py` simulator, the following function calls the APIs to create a federate:

```
def create_combo_federate(fedinitstring, name, period):
    fedinfo = h.helicsCreateFederateInfo()
    # "coreType": "zmq",
    h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
    h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
    # "loglevel": 1,
    h.helicsFederateInfoSetIntegerProperty(fedinfo, h.helics_property_int_log_level, 7)
    # "period": 60,
    h.helicsFederateInfoSetTimeProperty(fedinfo, h.helics_property_time_period, period)
    # "uninterruptible": false,
    h.helicsFederateInfoSetFlagOption(fedinfo, h.helics_flag_uninterruptible, False)
    # "terminate_on_error": true,
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
    # "name": "Charger",
    fed = h.helicsCreateCombinationFederate(name, fedinfo)
    return fed
```


The interface configurations are finalized and registered in one step using the following APIs:

```

fedinitstring = " --federates=1"
name = "Charger"
period = 60
fed = create_combo_federate(fedinitstring, name, period)

num_EVs = 5
end_count = num_EVs
endidx = {}
for i in range(0, end_count):
    end_name = f"Charger/EV{i+1}.so"
    endidx[i] = h.helicsFederateRegisterGlobalEndpoint(fed, end_name, "double")
    dest_name = f"Controller/ep"
    h.helicsEndpointSetDefaultDestination(endidx[i], dest_name)

pub_count = num_EVs
pubidx = {}
for i in range(0, pub_count):
    pub_name = f"Charger/EV{i+1}_voltage"
    pubidx[i] = h.helicsFederateRegisterGlobalTypePublication(
        fed, pub_name, "double", "V"
    )

sub_count = num_EVs
subidx = {}
for i in range(0, sub_count):
    sub_name = f"Battery/EV{i+1}_current"
    subidx[i] = h.helicsFederateRegisterSubscription(fed, sub_name, "A")

```

PyHELICS API configuration explanation

All the API calls reference the PyHELICS library with

```
import helics as h
```

Federate Creation `create_combo_federate()`

- **`h.helicsCreateFederateInfo()`** - Sets the federate information variable (set to `fedinfo`)
- **`h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")`** - Sets the core type for `fedinfo` to `zmq`
- **`h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)`** - Sets the number of federates (`fedinitstring` has been passed as " --federates=1")
- **`h.helicsFederateInfoSetIntegerProperty()`** - Sets log level calling another API, `h.helics_property_int_log_level`
- **`h.helicsFederateInfoSetTimeProperty()`** - Sets time information. This API must receive another API to distinguish which type of time property to set. The period is set with `h.helics_property_time_period`, and period has been pass to this function

- **h.helicsFederateInfoSetFlagOption()** - API to set a flag for the federate. The flag we are setting is `h.helics_flag_uninterruptible` to `False`, to mirror the JSON configuration
- **h.helicsFederateInfoSetFlagOption()** - API to set a flag for the federate. The flag we are setting is `h.HELICS_FLAG_TERMINATE_ON_ERROR` to `True`
- **fed = h.helicsCreateCombinationFederate(name, fedinfo)** - Creates the combination federate with the name passed to this function (Charger) and the information set above for `fedinfo`

Federate Interface Configuration and Registration

- **Endpoints**
 - **h.helicsFederateRegisterGlobalEndpoint(fed, end_name, 'double')** - The fed has been created, `end_name` is set in a loop, and the endpoint is registered as global double. This API registers the id object for each endpoint, `endid[i]`
 - **h.helicsEndpointSetDefaultDestination(endid[i], dest_name)** - As with the JSON configuration, a default destination is set with a destination name, `'Controller/ep'`, for each endpoint object
- **Publications**
 - **h.helicsFederateRegisterGlobalTypePublication(fed, pub_name, 'double', 'V')** - The publication interfaces are registered for the `fed` by looping through `pub_name`. The interface is given a datatype of `double`, units of `V` for volts, and designated as global type
- **Subscriptions**
 - **h.helicsFederateRegisterSubscription(fed, sub_name, 'A')** - The subscription interfaces are registered for the `fed` by looping through `sub_name`. The interface is given units of `A` for amps. Alternatively, the PyHELICS API for Inputs can be used: **h.helicsFederateRegisterGlobalTypeInput(fed, sub_name, 'double', 'A')**

Interface configuration, including federate creation and registration, is done prior to the co-simulation execution. The next section in this User Guide places federate interface configuration in the context of the co-simulation stages and discusses the four stages of the co-simulation.

Timing Configuration

The two fundamental roles of a co-simulation platform are to provide a means of data exchange between members of the co-simulation (federates) and a means of keeping the federation synchronized in simulated time.

In HELICS, time synchronization across the federates is managed by each federate requesting a time (via a HELICS API call). When granted a time, the federate generally does the following:

1. Checks all its inputs and grab any new signals (values or messages) that have been sent to it,
2. Execute its native simulation code and update its state to the current simulated time (*e.g.* solving equations governing the behavior of a physical model, calculating a control action, processing and logging data, etc),
3. Publish new values or send new messages to other federates, and
4. Request the next simulated time to update again.

Sometimes this timing configuration is determined by the construction of the simulator (for example, if it has a fixed simulation time step size) and sometimes the simulator will have nothing to do until it receives a new input (for example, with a controller).

In most federates there will be a line of code that look like this (at least if their *using the C API*):

```
t = h.helicsFederateRequestTime(fed, time_requested)
```

It is the role of each federate to determine which time it should request and it is the job of those integrating the simulator with HELICS to determine how best to estimate that value. For some simulators, `time_requested` will be the current simulated time (`t`) plus a time step. For other simulators, `time_requested` may be the final time (`HELICS_TIME_MAXTIME`) (see the example on [Combination Federates](#) for more details on this), and it will only be granted time (interrupted, configured as `"uninterruptible": false`) when there are relevant updates provided by other federates in the co-simulation. Generally, time requests are blocking calls and our federate will do nothing until the HELICS core has granted a time to it.

When a federate makes a time request it calls a HELICS function that blocks the execution of that thread in HELICS. (If the simulator in question is multi-threaded then other threads can continue to operate; hopefully whatever they're working on is largely independent of the rest of the federation.) The federate sits and waits for a return value from that function (the granted time), allowing the rest of the federation to execute. The implication of making a time request is that, given the current state of its boundary conditions, the federate has no tasks to execute until the time it is requesting, or until it receives from another federate a new value that changes its boundary conditions.

After making a time request, federates are granted a time by their HELICS core and the time they are granted will be one of two values: the time they requested (or the next available valid time as defined by their configuration) or an earlier valid time. Being granted a time earlier than requested is always accompanied by a new value or message in one of its inputs, subscriptions, or endpoints. A change in the federate's boundary conditions may require a change in one of the outputs for that federate and its core is obliged to wake up the federate so it can process this new information.

Based on the time requests and grants from all the connected federates, a core will determine the next time it can grant to a federate to guarantee none of the federates will be asked to simulate a point in time that occurs in the past. Every federate will receive a time that is the same as or larger than the last time it was granted. HELICS does support a configuration and some other situations that allows a federate to break this rule, but this is a very special situation and would require all the federates to support this jumping back in time, or accept non-causality and some randomness in execution.

The [section on federates](#) addressed the data-exchange responsibility of the co-simulation platform and this will address the timing and synchronization requirements. These two functions work hand-in-hand; for data-exchange between federates to be productive, data must be delivered to each federate at the appropriate time.

HELICS co-simulations end under one of two conditions: when all federates have been granted the time of `HELICS_TIME_MAXTIME` or when all federates have notified the broker (via their core) that they are terminating. The termination of the federates triggers a cascade of terminations throughout the federation: once all the federates associated with a core have terminated, the core itself terminates and once all cores associated with a broker have terminated, the broker itself terminates. This concludes the co-simulation and leaves the original models, configuration files, executing simulators, and results files in place for review.

Timing Configuration Options

Managing the timing of federate co-simulation is one of the most important and often challenging aspects of co-simulation. It is not uncommon for a federation to require that certain federates run at particular times or after certain other federates. HELICS provides a wide variety of timing parameters that can be configured for each federate (see the "Timing" section of the [Configuration Options Reference](#)).

The same JSON configuration file used to set the publications, subscriptions, and endpoints (as discussed in the [section on federates](#)) also controls how the federate manages its timing within the co-simulation.

Below is an example of how the most common of these are implemented in a federate configuration JSON file:

```
{
  "name": "generic_federate",
```

(continues on next page)

(continued from previous page)

```
...
"period": 1.0,
"offset": 1.0,
"time_delta": 10.0,
"uninterruptible": false,
"wait_for_current_time_update": true,
...
}
```

period, offset, and time_delta are all related and defined with units of seconds.

- **period**: Defines the resolution of the federate and is often tied to the underlying simulation tool. Period forces time grants to specific intervals.
- **offset**: Requires all time grants to be offset in time from the intervals defined by period by the amount indicated.
- **time_delta**: Forces the granted time to a minimum interval from the last granted time.

The granted time will be of value $n \times \text{period} + \text{offset}$ and it must be later than the last grant by time time_delta.

The other two options are common flags which may be invoked:

- **uninterruptible**: Forces the granted time to be the requested time. Generally HELICS will grant a federate a time when it receives new values on any of its inputs under the assumption that the federate is inherently interested in responding to new information to which it has subscribed. If that is not the case, setting this flag will reduce nuisance grants and move the federate forward in a predictable manner.
- **wait_for_current_time_update**: Force the federate with this flag set to be the last one granted a given time and thereby ensures that all other federates have produced outputs for that time. By being last, the federate in question will have updated outputs from all other federates and have the most comprehensive understanding of the system state at that simulated time.

Example: Timing in a Small Federation

For the purposes of illustration, let's suppose that a co-simulation federation with the following timing parameters has been assembled:

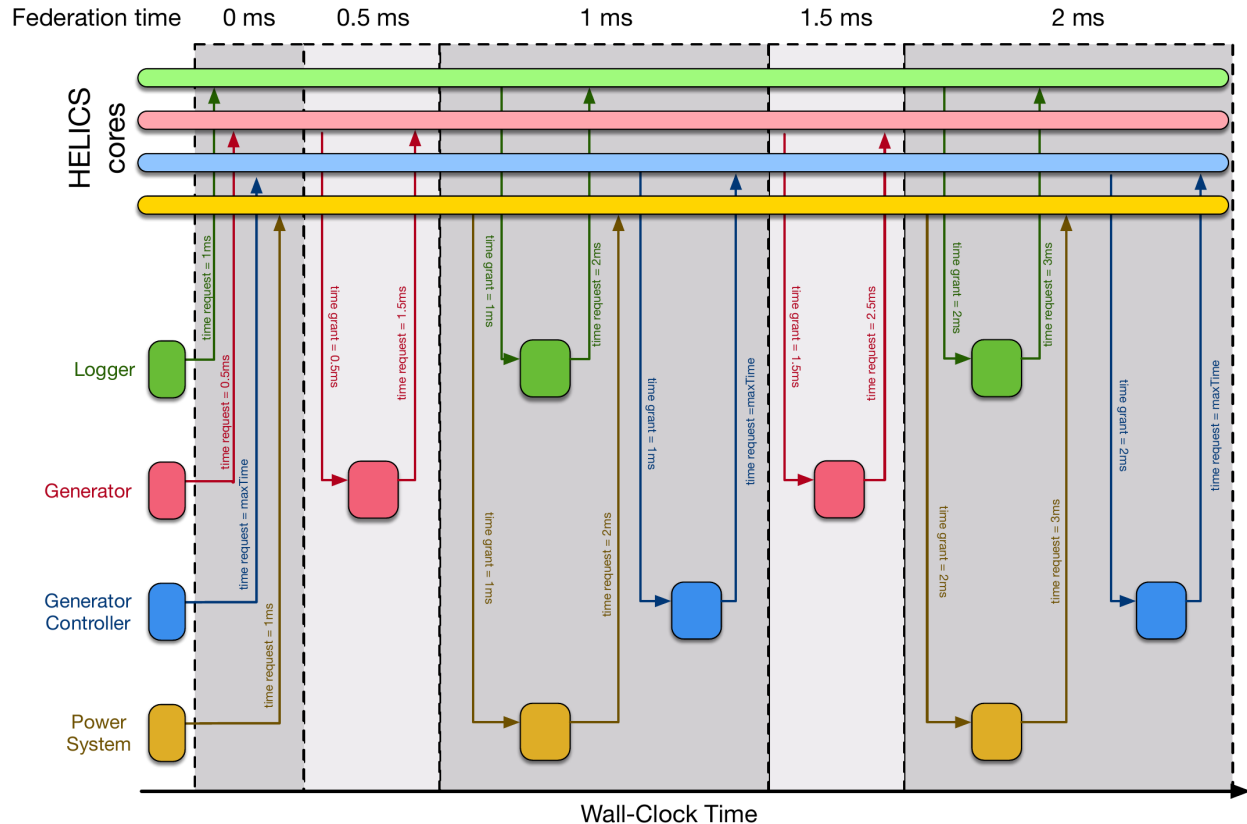
- **Logger** - This federate is a results logger and simply writes out to files the current values of various publications made by the other federates in the co-simulation. This logging simulator will record values every 1 ms and as such, the JSON config sets period to this value and sets the uninterruptible flag.
- **Generator** - This is a generator simulator that specializes in comprehensive modeling of the machine dynamics. The Generator will have an endpoint used to receive commands from the Generator Controller and subscriptions to outputs from the Power System that provide inputs necessary to calculate its internal dynamics.

The models of the generator are valid at a time-step of 0.1 ms and thus the simulator integrator requires that the period of the HELICS interface be set to some multiple of 0.1. In this case we'll use 1 ms and to ease integration with the Power System federate, it will also have an offset of 0.5 ms.

- **Generator Controller** - This is an event-based simulator, updating the control commands to the Generator federate whenever new inputs are received from the Power System federate (subscriptions to the physical values it calculates). As such, it will always request HELICS_TIME_MAXTIME, expecting to be granted times whenever the state of the Power System federate changes. The timeDelta will be set to 0.010 ms to replicate the time it takes to calculate and communicate the command signals to the Generator.

- **Power System** - This federate is a classic power system dynamics simulator with a fixed time-step of 1 ms. The integrator of this simulator choose to realize this by setting the `uninterruptible` flag and hard-coding the time requests to advance at 1 ms intervals.

Below is a timing diagram showing how these federates interact during a co-simulation. The filled blocks show when each federate has been woken up and is active.



Items of notes:

- Generator Controller gets granted a time of 1 ms (at the first grant time) even though is requested `HELICS_TIME_MAXTIME` because a message was created by the Power System federate at that time stamp. As Generator Controller depends on nothing else, HELICS was able to grant it the same time as Power System even though it is clearly performing its calculations after Power System has performed its.
- Relatedly, Generator Controller requests a time of `HELICS_TIME_MAXTIME` once it has calculated the new control signals for Generator. Due to the value set by `timeDelta`, the soonest time it can be granted would be 0.01 ms after its most recent granted time (1.01 in the case of the first operational period, 2.01 in the case of the second period.)
- When Logger is granted a time of 1 ms, the values it will record are those previously published by other federates. Specifically, the new values that Power System is calculating are not available for Logger to record.

Co-simulation Stages

HELICS has several stages to the co-simulation. Creation, initialization, execution, and final state. A call to `helicsFederateEnterExecutingMode()` is the transition between initialization and execution.

- *Creation*
 - *Registration*
 - * *Using a JSON Config File*
 - * *Using PyHELICS API Calls*
 - *Collecting the Interface Objects*
- *Initialization*
- *Execution*
 - *Get Inputs*
 - *Internal Updates and Calculations*
 - *Publish Outputs*
- *Final State*

Creation

For the purposes of these examples, we will assume the use of a Python binding. If, as the simulator integrator, you have needs beyond what is discussed here you'll have to dig into the [developer documentation on the APIs](#) to get the details you need.

To begin, at the top of your Python module (*after installing the Python HELICS module*), you'll have to import the HELICS library, which will look something like this:

```
import helics as h
```

Registration

As discussed in the previous section on [Federate Interface Configuration](#), configuration of federates can be done with either JSON config files or with the simulator's API.

Using a JSON Config File

In HELICS there is a single API call that can be used to read in all of the necessary information for creating a federate from a JSON configuration file. The JSON configuration file, as discussed earlier in this guide, contains both the federate info as well as the metadata required to define the federate's publications, subscriptions and endpoints. The API calls for creating each type of federate are given below.

For a value federate:

```
fed = h.helicsCreateValueFederateFromConfig("fed_config.json")
```

For a message federate:

```
fed = h.helicsCreateMessageFederateFromConfig("fed_config.json")
```

For a combination federate:

```
fed = h.helicsCreateCombinationFederateFromConfig("fed_config.json")
```

In all instances, this function returns the federate object `fed` and requires a path to the JSON configuration file as an input.

Using PyHELICS API Calls

Additionally, there are ways to create and configure the federate directly through HELICS API calls, which may be appropriate in some instances. First, you need to create the federate info object, which will later be used to create the federate:

```
fed = h.helicsCreateFederateInfo()
```

Once the federate info object exists, HELICS API calls can be used to set the *configuration parameters* as appropriate. For example, to set the `only_transmit_on_change` flag to true, you would use the following API call:

```
h.helicsFederateInfoSetFlagOption(fed, 6, True)
```

(The “6” there is the integer value for appropriate HELICS enumeration. The definition of the enumerations can be found in the [C++ API reference](#) and also cross shown in the [Configurations Options Reference](#).)

Once the federate info object has been created and the appropriate options have been set, the helics federate can be created by passing in a unique federate name and the federate info object into the appropriate HELICS API call. For creating a value federate, that would look like this:

```
fed = h.helicsCreateValueFederate(federate_name, fi)
```

Once the federate is created, you now need to define all of its publications, subscriptions and endpoints. The first step is to create them by registering them with the federate with an API call that looks like this:

```
pub = h.helicsFederateRegisterPublication(fed, key, data_type)
```

This call takes in the federate object, a string containing the publication key (which will be prepended with the federate name), and the data type of the publication. It returns the publication object. Once the publication, subscription and endpoints are registered, additional API calls can be used to set the info field in these objects and to set certain options. For example, to set the only transmit on change option for a specific publication, this API call would be used:

```
pub = h.helicsPublicationSetOption(pub, 454, True)
```

Once the federate is created, you also have the option to set the federate information at that point, which - while functionally identical to setting the federate info in either the federate config file or in the federate info object - provides integrators with additional flexibility, which can be useful particularly if some settings need to be changed dynamically during the cosimulation. The API calls are syntactically very similar to the API calls for setting up the federate info object, except instead they target the federate itself. For example, to revisit the above example where the `only_transmit_on_change` flag is set to true in the federate info object, if operating on an existing federate, that call would be:

```
h.helicsFederateSetFlagOption(fed, 6, True)
```

Collecting the Interface Objects

Having configured the publications, subscriptions and endpoints and registered this information with HELICS, the channels for sending and receiving this information have been created within the cosimulation framework. If you registered the publication, subscriptions and endpoints within your code (i.e., using HELICS API calls), you already have access to each respective object as it was returned when made the registration call. However, if you created your federate using a configuration file which contained all of this information, you now need to retrieve these objects from HELICS so that you can invoke them during the execution of your cosimulation. The following calls will allow you to query HELICS for the metadata associated with each publication. Similar calls can be used to get input (or subscription) and endpoint information.

```
pub_count = h.helicsFederateGetPublicationCount(fed)
pub = h.helicsFederateGetPublicationByIndex(fed, index)
pub_key = h.helicsPublicationGetKey(pub)
```

The object returned when the `helicsFederateGetPublicationByIndex()` method is invoked is the handle used for retrieving other publication metadata (as in the `helicsPublicationGetKey()` method) and when publishing data to HELICS (as described in the execution section below).

Initialization

Initialization mode exists to help a federation reach a consistent state or otherwise generally prepare to begin the advancement through time. Each federate can call `helicsFederateEnterInitializingMode()` and perform whatever internal set-up it needs to do as well as publish outputs that will be available to the rest of the federation at simulation time $t=0$ when entering execution mode (see the next section).

If the federation needs to iterate in initialization mode prior to entering execution mode each federate calls `helicsFederateEnterExecutingModeIterative()`. This API has two special aspects:

1. Calling the API requires that the federate declare its needs for iteration using an enumeration:

`NO_ITERATION` – don't iterate

`FORCE_ITERATION` – guaranteed iteration, stay in iteration mode

`ITERATE_IF_NEEDED` – If there is data available from other federates Helics will iterate, if no additional data is available it will move to execution mode and have granted time=0.

2. The API returns an enumeration indicating the federation's iteration state:

`NEXT_STEP` - Iteration has completed and the federation should move to the next time step. In the case of exiting initialization, this will be the time between $t=0$ (which was just completed by the iteration process) and the next time grant.

`ITERATING` - Federation has not ceased iterating and will iterate once again. During this time the federate will need to check all its inputs and subscriptions, recalculate its model, and produce new outputs for the rest of the federation.

To implement this initialization iteration, all federates need to implement a loop where `helicsFederateEnterExecutingModeIterative()` is repeatedly called and the output of the call is evaluated. The call to the API needs to use the federate's internal evaluation of the stability of the solution to determine if needs to request another iteration. The returned value of the API will determine whether the federate needs to re-solve its model with new inputs from the of the federation or enter normal execution mode where it can enter execution mode.

Execution

Once the federate has been created, all subscriptions, publications and endpoints have been registered and the federation initial state has been appropriately set, it is time to enter execution mode. This can be done with the following API call:

```
h.helicsFederateEnterExecutingMode(fed)
```

This method call is a blocking call; your custom federate will sit there and do nothing until all other federates have also finished any set-up work and have also requested to begin execution of the co-simulation. Once this method returns, the federation is effectively at simulation time of zero.

At this point, each federate will now set through time, exchanging values with other federates in the cosimulation as appropriate. This will be implemented in a loop where each federate will go through a set of prescribed steps at each time step. At the beginning of the cosimulation, time is at the zeroth time step ($t = 0$). Let's assume that the cosimulation will end at a pre-determined time, $t = \text{max_time}$. The nature of the simulator will dictate how the time loop is handled. However, it is likely that the cosimulation loop will start with something like this:

```
t = 0
while t < end_time:
    pass # cosimulation code would go here
```

Now, the federate begins to step through time. For the purposes of this example, we will assume that during every time step, the federate will first take inputs in from the rest of the cosimulation, then make internal updates and calculations and finish the time step by publishing values back to the rest of the cosimulation before requesting the next time step.

Get Inputs

The federate will first listen on each of its inputs (or subscriptions) and endpoints to see whether new information has been sent from the rest of the federation. The first code sample below shows how information can be retrieved from an input (or subscriptions) through HELICS API calls by passing in the subscription object. As can be seen, HELICS has built in type conversion (*where possible*) and regardless of how the sender of the data has formatted it, HELICS can present it as requested by the appropriate method call.

```
int_value = h.helicsInputGetInteger(sub)
float_value = h.helicsInputGetDouble(sub)
real_value, imag_value = h.helicsInputGetComplex(sub)
string_value = h.helicsInputGetChar(sub)
...
```

It may also be worth noting that it is possible on receipt to check whether an input has been updated before retrieving values. That can be done using the following call:

```
updated = h.helicsInputIsUpdated(sid)
```

Which returns true if the value has been updated and false if it has not.

Receiving messages at an endpoint works a little bit differently than when receiving values through a subscription. Most fundamentally, there may be multiple messages queued at an endpoint while there will only ever be one value received at a subscription (the last value if more than one is sent prior to being retrieved). To receive all of the messages at an endpoint, they needed to be popped off the queue. An example of how this might be done is given below.

```
while h.helicsEndpointPendingMessages(end) > 0:
    msg_obj = h.helicsEndpointGetMessageObject(end)
```

To get the source of each of the messages received at an endpoint, the following call can be used:

```
msg_source = h.helicsMessageGetOriginalSource(msg_obj)
```

Internal Updates and Calculations

At this point, your federate has received all of its input information from the other federates in the co-simulation and is now ready to run whatever updates or calculations it needs to for the current time step.

Publish Outputs

Once the new inputs have been collected and all necessary calculations made, the federate can publish whatever information it needs to for the rest of the federation to use. The code sample below shows how these output values can be published out to the federation using HELICS API calls. As in when reading in new values, these output values can be published as a variety of data types and HELICS can handle type conversion if one of the receivers of the value asks for it in a type different than published.

```
h.helicsPublicationPublishInteger(pub, int_value)
h.helicsPublicationPublishDouble(pub, float_value)
h.helicsPublicationPublishComplex(pub, real_value, imag_value)
h.helicsPublicationPublishChar(pub, string_value)
...
```

For sending a message through an endpoint, that once again looks a little bit different, in this case because - unlike with a publication - a message requires a destination. If a default destination was set when the endpoint was registered (either through the config file or through calling `h.helicsEndpointSetDefaultDestination()`), then an empty string can be passed. Otherwise, the destination must be provided as shown in API call below where `dest` is the destination and `msg` is the message to be sent.

```
h.helicsEndpointSendMessageRaw(end, dest, msg)
```

Final State

Once the federate has completed its contribution to the co-simulation, it needs to close out its connection to the federation. Typically a federate knows it has reached the end of the co-simulation when it is granted `maxTime`. To leave the federation cleanly (without causing errors for itself or others in the co-simulation) the following process needs to be followed:

```
h.helicsFederateFinalize(fed)
h.helicsFederateFree(fed)
h.helicsCloseLibrary()
```

`helicsFederateFinalize()` signals to the core and brokers that this federate is leaving the co-simulation. This process will take an indeterminate amount of time and thus it is necessary to poll the connection status to the broker. Once that connection has closed, the memory of the federate (associated with HELICS) is freed up with `helicsFederateFree()` and the processes in the HELICS library are terminated with `helicsCloseLibrary()`. At this point, the federate can safely end execution completely.

Logging

Logging in HELICS provides a way to understand the operation of a federate and is normally handled through an independent thread. The thread prints message to the console and or to a file as events within a federate occur. This section discusses how to use the log files to confirm the co-simulation executed properly and to debug when it doesn't.

- *Log Levels*
- *Setting up the Simulator for Logging*
- *Setting up the Federate for Logging*
- *Setting up the Core/Broker for Logging*

Log Levels

There are several levels used inside HELICS for logging. The level can be set with the enumerations when using an API to set the logging level. When configuring the log level via an external JSON config, the enumerations are slightly different:

API enumeration	JSON config keyword
<code>HELICS_LOG_LEVEL_NO_PRINT</code>	<code>no_print</code>
<code>HELICS_LOG_LEVEL_ERROR</code>	<code>error</code>
<code>HELICS_LOG_LEVEL_WARNING</code>	<code>warning</code>
<code>HELICS_LOG_LEVEL_SUMMARY</code> <code>summary</code>	
<code>HELICS_LOG_LEVEL_CONNECTIONS</code> <code>connections</code>	
<code>HELICS_LOG_LEVEL_INTERFACES</code> <code>interfaces</code>	
<code>HELICS_LOG_LEVEL_TIMING</code> <code>timing</code>	
<code>HELICS_LOG_LEVEL_DATA</code> <code>data</code>	
<code>HELICS_LOG_LEVEL_TRACE</code> <code>trace</code>	

- `helics_log_level_no_print` Don't log anything
- `helics_log_level_error` Log error and faults from within HELICS
- `helics_log_level_warning` Log warning messages of things that might be incorrect or unusual
- `helics_log_level_summary` Log summary messages on startup and shutdown. The Broker will also generate a summary with the number of federates connected and a few other items of information
- `helics_log_level_connections` Log a message for each connection event (federate connection/disconnection)
- `helics_log_level_interfaces` Log messages when interfaces, such as endpoints, publications, and filters are created
- `helics_log_level_timing` Log messages related to timing information such as mode transition and time advancement
- `helics_log_level_data` Log messages related to data passage and information being sent or received
- `helics_log_level_trace` Log all internal message being sent

NOTE: these levels currently correspond to (-1 through 7) but this may change in future major version numbers to allow more fine grained control.

`timing`, `data` and `trace` log levels can generate a large number of messages and should primarily be used for debugging. `trace` will produce a very large number of messages most of which deal with internal communications and is primarily for debugging timing in HELICS.

Log lines will often look like

```
echo1 (131072) (0)::Time mismatch detected granted time >requested time 5.5 vs 5.0
```

or

```
commMessage|26516-enRPa-PzaBB-ZG190-lj14t:got new broker information
```

which includes a name and internal id code for the federate followed by a time in parenthesis and the message. If it is a warning or error, there will be an indicator before the object name. Names for brokers or cores are often auto generated and look like 26516-enRPa-PzaBB-ZG190-lj14t which is essentially a random string with a thread id in the front. In this case, the `commMessage` indicates it came from one of the communication modules.

Setting up the Simulator for Logging

The *Fundamental Base Example* incorporates simple logging for the two federate co-simulation. These federates, Battery and Charger, are user-written in Python with PyHELICS, so we have the luxury of setting up the simulator for logging.

In the Battery simulator, we need to import `logging` and set up the logger:

```
import logging

logger = logging.getLogger(__name__)
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

Now we can use the `logger` to print different levels of detail about the co-simulation execution to log files. These files will be generated for each federate in the co-simulation and the broker with the naming convention “name assigned to federate/broker”.log.

A set of functions are available for individual federates to generate log messages. These functions must be placed in the simulator. In the *Fundamental Base Example*, the `logger.info()` and `logger.debug()` methods are used. Stipulating different types of log messages allows the user to change the output of the log files in one location – the config file for the federate. These will log a message at the `log_level` specified in the config file.

```
logger.info("Only prints to log file if log_level = 2 or summary")
logger.debug("Only prints to log file if log_level = 6 or data")
logger.error("Only prints to log file if log_level = 0 or error")
logger.warning("Only prints to log file if log_level = 1 or warning")
```

Setting up the Federate for Logging

Most of the time the log for a federate is the same as for its core. This is managed through a few properties in the `HelicsFederateInfo` class which can also be directly specified through the property functions.

- `helics_property_int_log_level` - General logging level applicable to both file and console logs
- `helics_property_int_file_log_level` Level to log to the file
- `helics_property_int_console_log_level` Level to log to the console

These properties can be set using the JSON configuration for each federate:

```
{
  "name": "Battery",
  "log_level": 1,
  ...
}
```

Or with the API interface functions for each federate:

```
h.helicsFederateInfoSetIntegerProperty(fed, h.helics_property_int_log_level, 1)
```

Setting up the Core/Broker for Logging

It is possible to specify a log file to use on a core. This can be specified through the coreinit string `--logfile logfile.txt`

or on a core object

```
h.helicsCoreSetLogFile(core, "logfile.txt")
```

A similar function is available for a broker. The Federate version will set the logFile on the connected core.

With the API:

```
h.helicsFederateSetLogFile(fed, "logfile.txt")
```

Within the `helics_cli` runner JSON:

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 2 --loglevel=7",
      "host": "localhost",
      "name": "broker"
    },
    ...
  ],
  "name": "fundamental_default"
}
```

A federate also can set a logging callback so log messages can be processed in whatever fashion is desired by a federate.

In C++ the method on a federate is:

```
setLoggingCallback (const std::function<void(int, const std::string &, const std::string_
↵&)> &logFunction);
```

In PyHELICS:

```
h.helicsFederateSetLoggingCallback(fed, logger, user_data)
```

The callback take 3 parameters about a message and in the case of C callbacks a pointer to user data.

- loglevel an integer code describing the level of the message as described above.
- identifier a string with the name of the object generating the message (may be empty)

- message the actual message to log

helics_cli

Execution of the HELICS co-simulation is done from the command line with `helics_cli`, or Command Line Interface. Each simulator must be executed individually in order to join the federation. `helics_cli` condenses these command line execution commands into one executable, called the runner file.

All the *examples* are written with a runner file for execution. In the *Fundamental Base Example*, we need to launch three things: the broker, the Battery federate, and the Charger federate.

The file `fundamental_default_runner.json` includes:

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 2 --loglevel=7",
      "host": "localhost",
      "name": "broker"
    },
    {
      "directory": ".",
      "exec": "python -u Charger.py 1",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py 1",
      "host": "localhost",
      "name": "Battery"
    }
  ],
  "name": "fundamental_default"
}
```

This tells `helics_cli` to launch three federates named `broker`, `Charger`, and `Battery`. The `directory` tells HELICS the location of the executable. For the `broker`, the executable `helics_broker` should be *configured to suite your needs*. The `broker` is launched with the executable `helics_broker`, to which we pass the information `-f 2` meaning there will be two federates, and `--loglevel=7` meaning that we want *all internal messages* to be sent to the log file.

The other two federates are Python based. We instruct `helics_cli` to launch these with the `exec` command `python -u`.

Once the runner file is specified to include information about where the executables live (`directory`), the execution command for each (`exec`), the host, and the name, the entire federation can be launched with the following command:

```
> helics run --path=fundamental_default_runner.json
```

The next section discusses using the Web Interface to interact with a running HELICS co-simulation. The federation must be launched with `helics_cli` in order to use the Web Interface.

HELICS Web Interface

Once a federate has been granted the ability to move forward to a specific time (the granted time), the federate will execute its simulation, calculating its new state, behavior, or control action to advance to that time. Prior to these calculations, it will receive any messages that have been sent to it by other federates and after simulating up to the granted time, may send out messages with new values other federates may need.

Using the webserver that HELICS provides to access data about a cosimulation (without having to using the HELICS APIs yourself), HELICS also has a GUI via a web page that allows users to more easily run, monitor and debug a co-simulation.

Launching Web Interface

The web interface requires the use of *helics_cli* to run the co-simulation and is created by the following command:

```
$ helics_cli server  
127.0.0.1:8000
```

The response from *helics_cli* is the address of the web interface that can be used to run the co-simulation that has *helics_cli* has been configured to run. Copy and paste that into a web browser to access the interface.

Web Interface Overview

1 Load Configuration File

2 >> 3

Next Time Step: 4

5 Federate Configuration

Federate Name	Last Granted Time	Next Requested Time
Gridlab-d_1	3	4
Gridlab-d_2	3	4
Gridlab-d_3	3	4
Griddyn	3.1	2
hvac_controller_1	3	Inf
hvac_controller_2	3	Inf
hvac_controller_3	3	Inf
Gridlab-d_1	3	4
Gridlab-d_2	3	4
Gridlab-d_3	3	4

Showing 1 to 10 of 175 rows 10 rows per page

6 Publications

Key	Sender	Publication Time	Value	Value Updated
2	tempor	998	107.399462	true
0	occaecat	301	-73.754968	true
1	cillum	725	160.128881	true
3	deserunt	460	-125.855322	false

1. **Load web interface configuration** - A configuration file is provided to allow the user to be more specific about which federates and publications are of interest. Particularly for federations with large numbers of federates and/or large numbers of publications, it may not be desirable to display all of that information in the web interface, especially if it isn't needed. A configuration file can be loaded that will allow the user to restrict which federates and publications are added to the respective tables. xxxxxx - Do we need to advise users to limit the size of these tables for performance reasons. Or is this largely expected to be a convenience for them? At what point does size become a problem?
2. **Run co-simulation** - This button will launch the co-simulation via `helics_cli` and run it to completion. The values in the "Federate Configuration" and "Publication" fields will update periodically to reflect the current state of

the federation. This allows for some degree of monitoring while the co-simulation runs.

3. **Stop co-simulation** - Stops the currently running co-simulation, likely because it is clear something has gone horribly wrong.
4. **Debug run to next time grant** - Just pressing the play-pause button will advance the entire federation to the next granted time (across the entire federation). The text field updates with the next time to be granted but also allows the user to provide the granted time to run to next.
5. **Federate State** - Table showing the federates in the federation, the last simulated time they were granted and the simulated time they have requested. This list can be filtered using the “Search” text box though regular expressions are not supported.
6. **Publication** - Listing of publications from the federation showing the key, sending federate, simulated time of publication, value, and whether the value was updated at the last simulated time granted.

Web Interface Configuration File

Forthcoming description of how the web interface configuration file is formatted and how to edit it to get the right federates and publications to appear in the web interface tables.

Using the Web Interface

Normal, everyday runs

When you have a co-simulation that you’ve vetted and feel comfortable will run without difficulty (or at least you don’t anticipate a need to debug it), running the co-simulation to completion from the interface is straight-forward:

1. (optional) Load in a configuration file - If you just want to look at a few federates and/or publications to track the progress of the co-simulation, make a web interface configuration file that does so and load it up.
2. Press the “Run” button (number 2 in the screenshot above).

The `helics_cli` will launch the co-simulation and the web interface will update every xxxxxxxx seconds (or as specified in the configuration file). This periodically updating view is helpful if only to see where the co-simulation is at (simulation time-wise) but also helps confirm that message values are as expected. When the co-simulation is done, xxxxxxxx (as message will appear? How are we indicating this?). All messages from the co-simulation (or a subset as specified in the `helics_cli` configuration) have been stored in an SQLite database; the database can be directly queried or the data can be exported to file in a number of formats (CSV, JSON) and post-process by a tool of your choice.

Debugging runs

Just like writing code, it is not unusual for a co-simulation to not quite work write the first time it’s constructed and the web interface provides tooling to help verify and trouble-shoot the construction co-simulation. There’s a whole section on trouble-shooting techniques and we won’t re-iterate the guidance here but here’s how the web interface can be used try to get a diagnostic view of the federation’s operation.

1. (optional but highly recommended) Load in a configuration file - Limit the federates and publications to those of interest and most revealing of the state of the simulation.
2. Enter a simulated time of interest and press the “Run to next grant” button (number 4 in the above diagram)
3. Check the state of the federation - When the co-simulation stops at the indicated time, check that all federates of interest have been granted and are requesting expected times. Check the message table and verify that all messages of interest have published with reasonable/expected values.

4. As necessary, proceed to subsequent or later time steps to hunt down the particular problem of interest.

Web interface demonstration

Below is a demonstration of how the web interface is used to run Example xxxxxxxx.

The fundamental topics listed below cover the material necessary to build a fully functional co-simulation with HELICS, written for users who have little to no experience with co-simulation. Each section makes reference to the *Fundamental Examples* to allow the user to scaffold their learning with concrete and detailed examples. After working through the topics below, the user should be able to write their own simple co-simulation in Python with PyHELICS and understand how access resources improve the development of their co-simulations.

The topics considered “fundamental” to building a co-simulation with HELICS are:

- ***HELICS Terminology*** - Key terms and concepts to understand before running co-simulations with HELICS
- ***Federates*** - Discussion of the different types of federates in HELICS and how to configure them.
 - Value Federates
 - Message Federates
 - Filters
 - Value Types
- ***Federate Interface Configuration*** - How to connect an existing simulator with HELICS
 - *With JSON config file*
 - *With HELICS APIs*
- ***Timing Configuration*** - How HELICS coordinates the simulation time of all the federates in the federation
 - Timing Exercise
 - Timing Exercise answers
- ***Stages of the Co-simulation***
- ***Logging*** - Discussion of logging within HELICS and how to control it.
- ***Execution with helics_cli*** - The HELICS team has developed `helics_cli` as a standardized means of running HELICS co-simulations.
- ***Simulation Management*** - Using the webserver, HELICS also has a built-in web interface for running, monitoring, and diagnosing co-simulations.
- ***Simulator Integration*** - A guide for integrating HELICS into simulators.

1.3.3 Advanced Topics

Multi-Source Inputs

On occasion it is useful to allow multiple source to feed an input, creating an n -to-one relationship (publications to input). This could occur in situations like a summing junction, or a switch that can be turned on or off from multiple other federates. Alternatively, a multi-source input can be a convenient way to collecting multiple inputs into a vector for processing by the federate. While technically supported prior to 2.5.1 the control and access to this feature of HELICS was not well thought through or straightforward. The developments in 2.5.1 made it possible to specify a mathematical reduce operation on multiple inputs to allow access to them as a single value or vector.

Mechanics of multi-input handling

Internally, HELICS manages input data in a queue and when a federate is granted a time, the values are read and placed in a holding location by source. In many cases there is likely only to be a single source. But if multiple publications link to a single source the results are placed in a vector. The order of the values in that vector is determined by the order of linking when the federate with the multi-source input is created. If a single publication value is retrieved from the input, the newest value is given as if it were a single source. In case of ties (multiple publishing federates publishing values on the same timestep), the publication that connected first is given priority.

Controlling the behavior

A few flags are available to control or modify this behavior including limiting the number of connections and adjusting the priority of the different inputs sources. The behavior of inputs is controlled via flags using `helicsInputSetOption()` method.

The number of connections

There are several flags and handle options which can control this for Inputs

- `helics_handle_option_single_connection_only` : If set to true specified that an input may have only 1 connection
- `helics_handle_option_multiple_connections_allowed`: if set to true then multiple connections are allowed
- `helics_handle_option_connections`: takes an integer number of connections specifying that an input must have N number of connections or an error will be generated.

Controlling priority

The default priority of the inputs if they are published at the same time and only a single value is retrieved is in order of connection. This may not be desired so a few handle options are available to manipulate it.

- `helics_handle_option_input_priority_location` takes a value specifying the input source index to which it will give priority. If given multiple times it establishes an ordering of the input priority with the most recent call being highest priority. This allows signals that are received at the same time to be prioritized. For example, let's say the option is called first with a given value of "2" then again with a value of "1". If all signals are sent at the same simulated time, the source with index 1 will have highest priority, and in the case of a tie between sources 0 and 2, source 2 will have priority.
- `helics_handle_option_clear_priority_list` will erase the existing priority list.

Reduction operations on multiple inputs

The priority of the inputs is only applicable if the default operation to retrieve a single value is used. The option `helics_handle_option_multi_input_handling_method` can be used to specify a reduction operation on all the inputs to process them in some fashion a number of operations are available.

The handling method specifies how the reduction operation occurs the value can then be retrieved normally via any of the `getValue` methods on an input.

Configuration

Multi Input handling can be configured through the programming API or through a file based configuration.

C++

```
auto& in1 = vFed1->registerInput<double>("");
in1.addTarget("pub1");
in1.addTarget("pub2");
in1.addTarget("pub3");
in1.setOption(helics::defs::options::multi_input_handling_method,
              helics::multi_input_handling_method::average);
```

C

```
/*errors are ignored here*/
helics_input in1 = helicsFederateRegisterInput("",helics_data_type_double,"",nullptr);
helicsInputAddTarget(in1,"pub1",nullptr);
helicsInputAddTarget(in1,"pub2",nullptr);
helicsInputAddTarget(in1,"pub2",nullptr);
helicsInputSetOption(in1,helics_handle_option_multi_input_handling_method,helics_multi_
↪input_average_operation, nullptr);
```

Python

```
in1 = h.helicsFederateRegisterInput("", h.helics_data_type_double, "")
h.helicsInputAddTarget(in1, "pub1")
h.helicsInputAddTarget(in1, "pub2")
h.helicsInputAddTarget(in1, "pub2")
h.helicsInputSetOption(
    in1,
    helics_handle_option_multi_input_handling_method,
    helics_multi_input_average_operation,
)
```

The handling can also be configured in the configuration file for the federate

TOML

```
inputs=[
{key="ipt2", type="double", targets=["pub1","pub2"], connections=2, multi_input_
↪handling_method="average"}
]
```

JSON

```
"inputs": [
  {
    "key": "ipt2",
    "type": "double",
    "connections":2,
    "multi_input_handling_method":"average",
    "targets": ["pub1","pub2"]
```

(continues on next page)

(continued from previous page)

```
}  
]
```

The priority of the inputs in most cases determined by the order of adding the publications as a target. This is not strictly guaranteed to occur but is a general rule and only applies in the default case, and possibly the diff operation.

Example

An *explanation of a full co-simulation example* showing how a multi-source input might be used in a federation is provided in the [HELICS Examples repository](#).

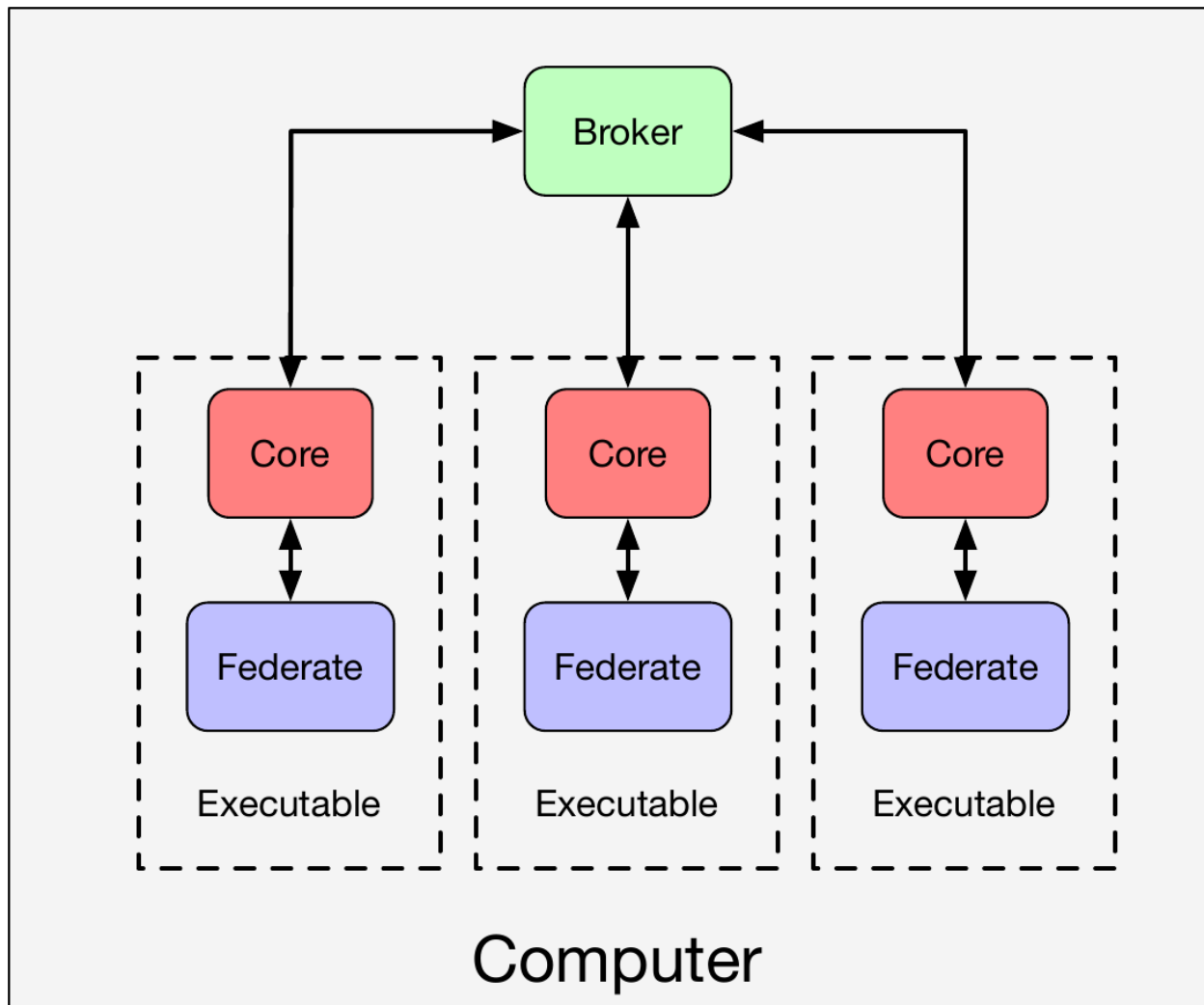
Co-simulation Architectures

There are several co-simulation architectures that can be constructed where the relationships between the federates, cores, and brokers can vary.

Simple Co-simulation

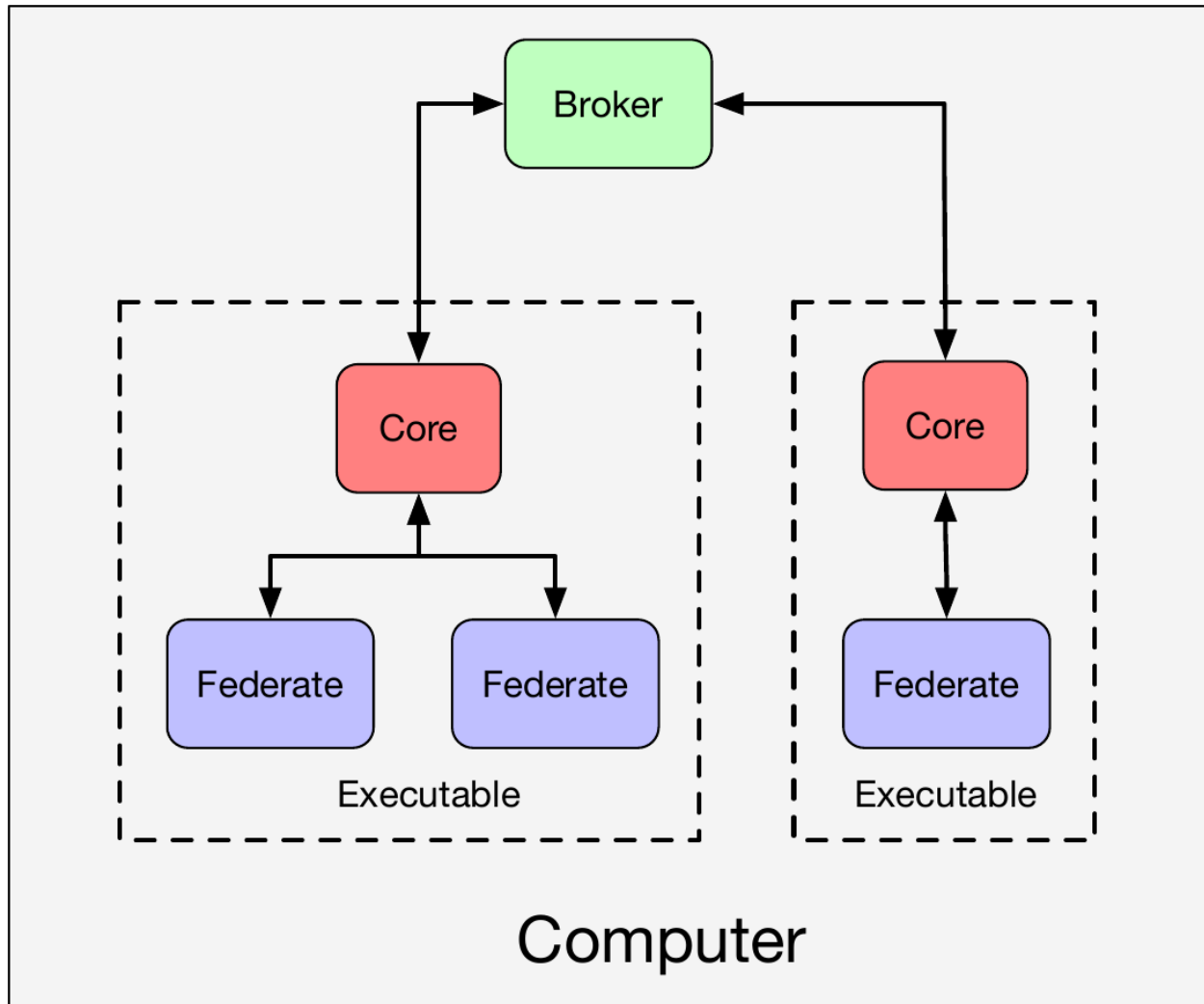
Broker topology is somewhat optional for simple co-simulations, but offers an increase in performance if it is possible to identify groups of federates that interact often with each other but rarely with the rest of the federation. In such cases, assigning that group of federates their own broker will remove the congestion their messages cause with the federation as a whole. The Fundamental Topics and Examples are built with a single broker.

The figure below shows the most common architecture for HELICS co-simulation. Each core has only one federate as an integrated executable, all executables reside on the same computer and are connected to the same broker. This architecture is particularly common for small federates and/or co-simulations under development. This is also the architecture for the *Fundamental Examples*.



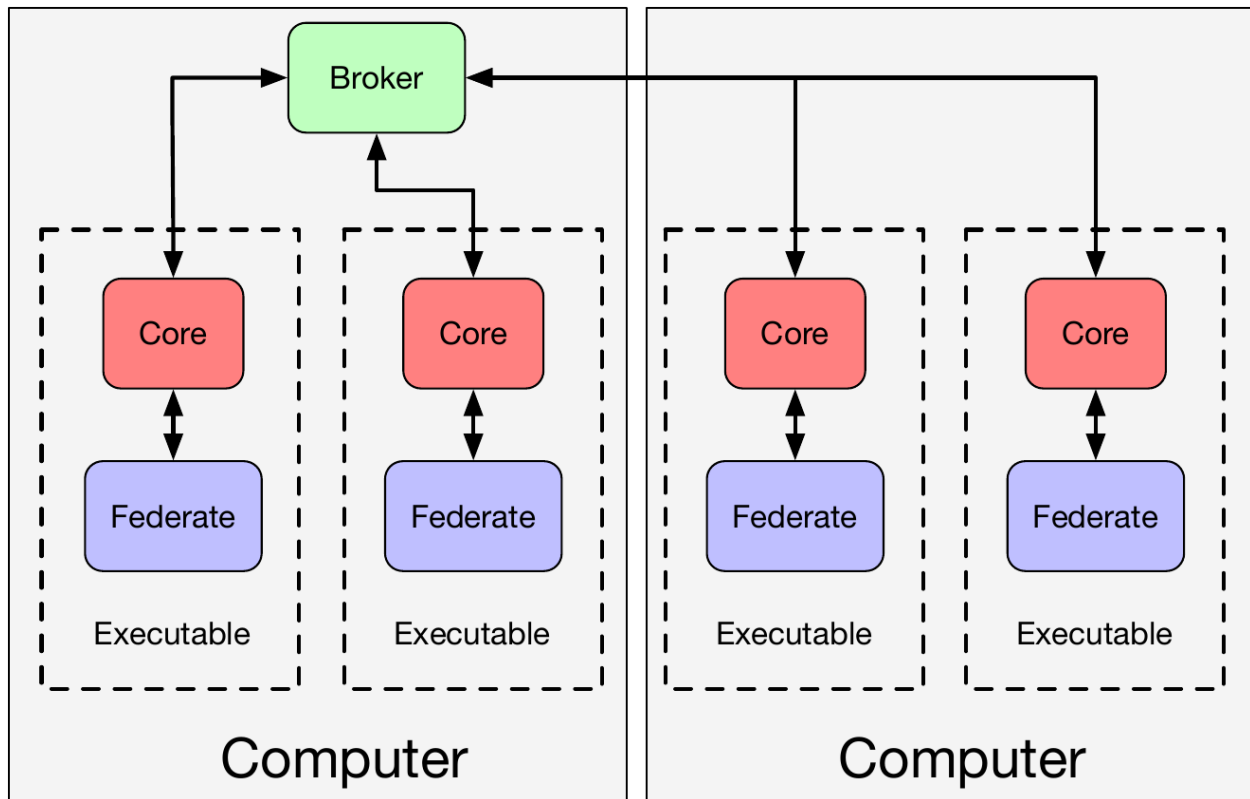
Multiple Federates on a Single Core

The architecture below shows a scenario where more than one federate is associated with a single core. For most simulators that have already been integrated with HELICS this architecture will generally not be used. For simulators that are multi-threaded by nature and typically represent multiple independent simulated entities (federates to HELICS), HELICS can be configured to facilitate message passing between threads. For a co-simulation that exists entirely within a single executable, this architecture will provide the highest performance. For example, if a large number of small controllers are written as a single, multi-threaded application (perhaps all the thermostats in a large commercial building are being managed by a centralized controller), particularly where there is communication between the federates, using a single core inside a single multi-threaded application (with typically one thread per federate) will provide the highest level of performance.



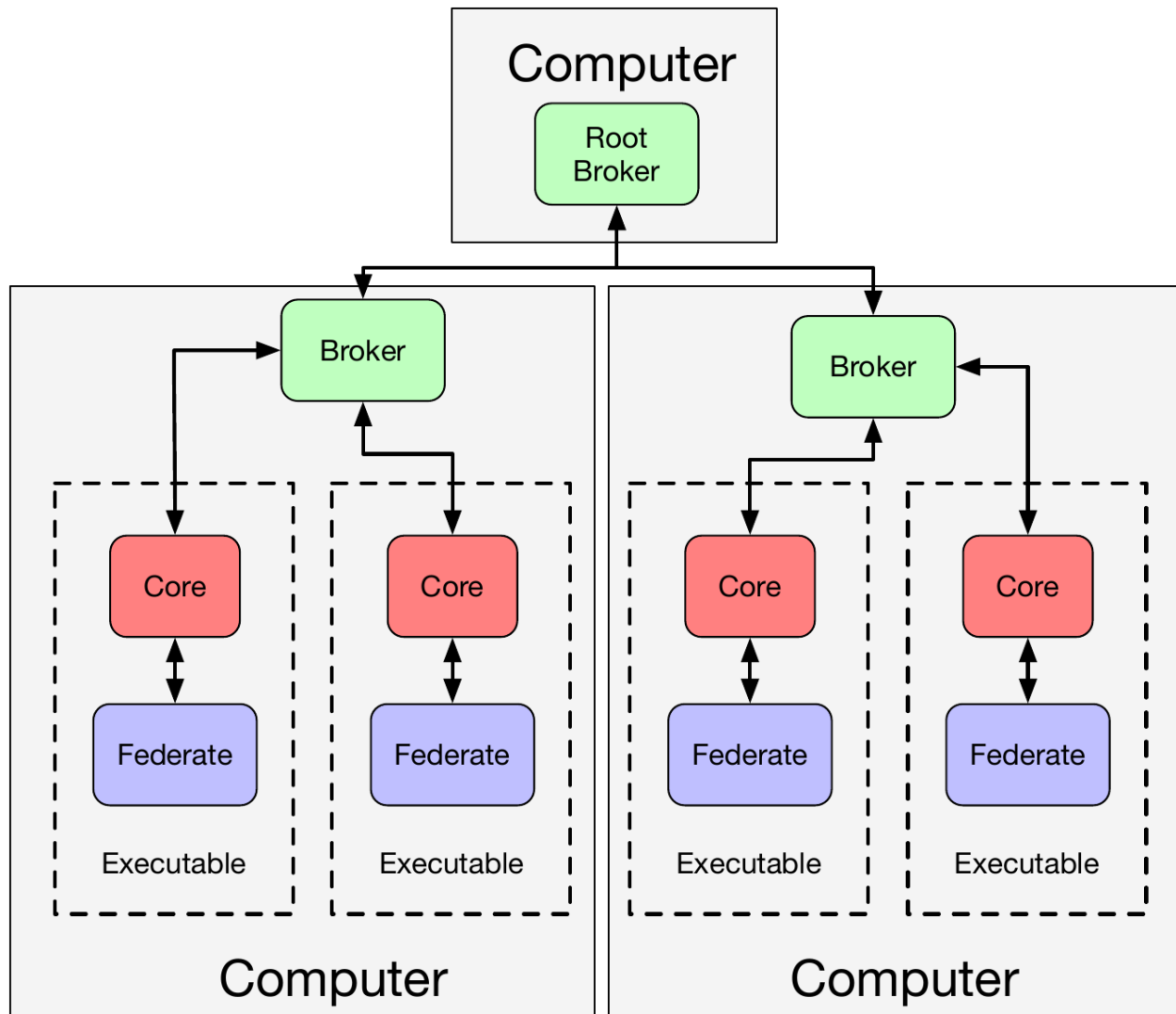
Computationally Heavy Federates

For co-simulations on limited hardware where a federate requires significant computational resources and high performance is important, it may be necessary to spread the federates out across a number of compute nodes to give each federate the resources it needs. All federates are still connected to a common broker and it would be required that the computers have a valid network connection so all federates can communicate with this broker. In this case, it may or may not be necessary to place the broker on its own compute node, based on the degree of competition for resources on its current compute node.



Multiple brokers

Alternatively, it would be possible to locate a broker on each computer and create a root broker on a third node. This kind of architecture could help if higher performance is needed and the federates on each computer primarily interact with each other and very little with the federates on the other computer. As compared to the previous architecture, adding the extra layer of brokers would keep local messages on the same compute node and reduce congestion on the root broker. An overview of how this is constructed is provided in the [section on broker hierarchies](#) and an example of this architecture (though running on a single compute node for demonstration purposes), is shown in the [broker hierarchy example](#)



Queries

Queries are an asynchronous means within a HELICS federation of asking for and receiving information from other federate components. A query provides the ability to evaluate the current state of a federation and typically addresses the configuration and architecture of the federation. Brokers, Federates, and Cores all have query functions. Federates are also able to define a callback for answering custom queries.

The general function looks like this:

C++

```
std::string query(const std::string& target, const std::string& queryStr)
```

Python

```
query_result = h.helicsCreateQuery(traget_string, query_string)
```

Targets

Each query must define a “target”, the component in the federation that is being queried. The target is either specified in terms of the relationship to the querying federate (*e.g.* “broker”, “core”) or by name of the federation component (*e.g.* “dist_system_1_fed”). The table below lists the valid query targets; if a federate happens to be named one of the target names listed below, it can not be queried by that name. For example, naming one of your brokers “broker” will prevent it being a valid target of a query by name. Instead, any federate that queries “broker” will end up targeting their broker.

target	Description
broker	The first broker encountered in the hierarchy from the caller
root, federation, rootbroker	The root broker of the federation
global	Retrieve the data associated with a global variable
parent	The parent of the caller
core	The core of a federate. This is not a valid target if called from a broker
federate	A query to the local federate or the first federate of a core
<object name>	any named object in the federation can also be queried, brokers, cores, and federates

Query String

The `queryStr` is the specific data being requested; the tables below show the valid data provided by each queryable federation component. All queries return a valid JSON string with invalid queries returning a JSON with an error code and error message. (The only exception is the `global_value` query which just returns a string containing global value.)

As of HELICS 2.7.0 Queries have an optional parameter to describe a sequencing mode. There are currently two modes, `HELICS_SEQUENCING_MODE_FAST` which travels along priority channels and is identical to previous versions in which all queries traveled along those channels. The other mode is `HELICS_SEQUENCING_MODE_ORDERED` which travels along lower priority channels but is ordered with all other messages in the system. This can be useful in some situations where you want previous messages to be acknowledged as part of the federation before the query is run. The `global_flush` query is forced to run in ordered mode at least until after it gets to the specified target.

Federate Queries

The following queries are defined for federates. Federates may specify a callback function which allows arbitrary user defined Queries. The queries defined here are available inside of HELICS.

The `global_time_debugging` and `global_flush` queries are also acknowledged by federates but it is not usually recommended to run those queries on a particular federate as they are more useful at higher levels. See the `Core` and `Broker` queries for more description of them. The difference between `tag/<tagname>` and `<tagname>` is that using the `tag/` prefix can retrieve any tag and will return an empty string if the tag doesn’t exist. Just using the tag name will not return tags of the same name as other queries and will generate an error response if the tag doesn’t exist.

Local Federate Queries

The following queries are defined for federates but can only be queried on the local federate, that is, the federate making the query. Federates may specify a callback function which allows arbitrary user defined Queries.

queryString	Description
updated_input_indices	vector of number of the inputs that have been updated [sv]
updated_input_names	names or targets of inputs that have been updated [sv]
updates	values of all currently updated inputs [structure]
values	current values of all inputs [structure]
time	the current granted time [string]

Core queries

The following queries will be answered by a core:

queryString	Description
name	the identifier of the core [string]
address	the network address of the core [string]
isinit	If the core has entered init mode [T/F]
isconnected	If the core has is connected to the network [T/F]
publications	current publications defined in a core [sv]
inputs	current named inputs defined in a core [sv]
endpoints	current endpoints defined in a core [sv]
filters	current filters of the core [sv]
federates	current federates defined in a core [sv]
dependenson	list of the objects this core depends on [sv]
dependents	list of dependent objects [sv]
dependencies	structure containing dependency information [structure]
federate_map	a Hierarchical map of the federates contained in a core [structure]
federation_state	a structure with the current known status of the brokers and federates [structure]
current_time	if a time is computed locally that time sequence is returned [structure]
global_time get	a structure with the current time status of all the federates/cores [structure]
current_state	The state of all the components of a core as known by the core [structure]
global_state	The state of all the components from the components [structure]
dependency_graph	a representation of the dependencies in the core and its federates [structure]
data_flow_graph	a representation of the data connections from all interfaces in a core [structure]
filtered_endpoints	data structure containing the filters on endpoints for the core[structure]
queries	list of dependent objects [sv]
version_all	data structure with the version string and the federates[structure]
version	the version string for the helics library [string]
counter	A single number with a code, changes indicate core changes [string]
global_time_debugging	return detailed time debugging state [structure]
global_flush	a query that just flushes the current system and returns the id's [structure]
tags	a JSON structure with the tags and values [structure]
tag/<tagname>	the value associated with a tagname [string]
<tagname>	the value associated with a tagname [string]

The version and version_all queries are valid but are not usually queried directly, but instead the same query is used on a broker and this query in the core is used as a building block.

Broker Queries

The following queries will be answered by a broker:

queryString	Description
name	the identifier of the broker [string]
address	the network address of the broker [string]
isinit	If the broker has entered init mode [T/F]
isconnected	If the broker is connected to the network [T/F]
publications	current publications known to a broker [sv]
endpoints	current endpoints known to a broker [sv]
federates	current federates under the brokers hierarchy [sv]
brokers	current cores/brokers connected to a broker [sv]
dependson	list of the objects this broker depends on [sv]
dependencies	structure containing dependency information for the broker [structure]
dependents	list of dependent objects [sv]
counts	a simple count of the number of brokers, federates, and handles [structure]
current_state	a structure with the current known status of the brokers and federates [structure]
global_state	a structure with the current state all system components [structure]
status	a structure with the current known status (true if connected) of the broker [structure]
current_time	if a time is computed locally that time sequence is returned, otherwise #na [string]
global_time	get a structure with the current time status of all the federates/cores [structure]
federate_map	a Hierarchical map of the federates contained in a broker [structure]
dependency_graph	a representation of the dependencies connections in all objects connected to a broker [structure]
data_flow_graph	a representation of the data connections from all interfaces in a federation [structure]
queries	list of dependent objects [sv]
version_all	data structure with the version strings of all broker components [structure]
version	the version string for the helics library [string]
counter	A single number with a code, changes indicate federation changes [string]
global_time_debugging	return detailed time debugging state [structure]
global_flush	a query that just flushes the current system and returns the id's [structure]
global_status	an aggregate query that returns a combo of global_time and current_state [structure]

`federate_map`, `dependency_graph`, `global_time`, `global_state`, `global_time_debugging`, and `data_flow_graph` when called with the root broker as a target will generate a JSON string containing the entire structure of the federation. This can take some time to assemble since all members must be queried. `global_flush` will also force the entire structure along the ordered path which can be quite a bit slower. Error codes returned by the query follow [http error codes](#) for “Not Found (404)” or “Resource Not Available (400)” or “Server Failure (500)”.

Usage Notes

Queries that must traverse the network travel along priority paths unless specified otherwise with a sequencing mode. The calls are blocking, but they do not wait for time advancement from any federate and take priority over regular communication.

The difference between `current_state` and `global_state` is that `current_state` is generated by information contained in the component so doesn't generate secondary queries of other components. Whereas `global_state` will reach out to the other components to get up to date information on the state.

Error Handling

Queries that can't be processed or are not recognized return a JSON error structure. The structure will contain an error code and message such as:

```
{
  "error": {
    "code": 404,
    "message": "target not found"
  }
}
```

The error codes match with [HTTP error codes](#) to the extent possible.

Application API

There are two basic calls in the application API as part of a [federate object](#). In addition to the call described above a second version omits the “target” specification and always queries the local federate.

```
std::string    query(const std::string& queryStr)
```

There is also an asynchronous version (that is, non-blocking) that returns a `query_id_t` that can be used in `queryComplete` and `isQueryComplete` functions.

```
query_id_t    queryAsync(const std::string& target, const std::string& queryStr)
```

In the header `<helics\queryFunctions.hpp>` a few helper functions are defined to vectorize query results and some utility functions to wait for a federate to enter init, or wait for a federate to join the federation.

C API and interface API's

Queries in the [C API](#) have the same valid targets and properties that can be queried but the construction of the query is slightly different. The basic operation is to create a query using `helicsQueryCreate(target, query)`. Once created, the target or query string can be changed with `helicsQuerySetTarget()` and `helicsQuerySetQueryString()`, respectively.

This function returns a query object that can be used in one of the execute functions (`helicsQueryExecute()`, `helicsQueryExecuteAsync()`, `helicsQueryBrokerExecute()`, `helicsQueryCoreExecute()`), to perform the query and receive back results. The query can be called asynchronously on a federate. The target field may be empty if the query is intended to be used on a local federate, in which case the target is assumed to be the federate itself. A query must be freed after use `helicsQueryFree()`.

Timeouts

As long as timeouts are enabled in the library itself, queries have a timeout system so they don't block forever if a federate fails or some other condition occurs. The current default is 15 seconds. It can be changed by using the command line option `--querytimeout` on cores or brokers (or in `--coreinitstring` on cores). In a later version an ability to set this and some other timeout values through properties will likely be added (HELICS 3.1). If the query times out a value of `#timeout` will be returned in the string.

Example

A full co-simulation example showing how queries can be used for *dynamic configuration* can be found [here](#) (with the source code in the [HELICS Examples repository](#)).

Interacting with a Running Simulation

Starting in HELICS 2.4 there is a webserver that can be run with the `helics_broker` or `helics_broker_server`. This requires using a boost version ≥ 1.70 . The Webserver can be disabled by the `HELICS_DISABLE_BOOST=ON` or `HELICS_DISABLE_WEBSERVER=ON` options being set.

Startup

The webserver can be started with the option `--http` to start a restful interface using HTTP, or `--web` to start a server using websockets. For example to run a broker server with `zmq` and the webserver active for 30 minutes, you can use the following:

```
helics_broker_server --http --zmq --duration 30minutes
```

The `--duration` is optional and the default is 30 minutes but any time can be specified.

The web server is configured by default on the localhost address port 80. If you want to configure this it can be done through a configuration file. The format is json.

```
{
  "http": {
    "port": 8080,
    "interface": "0.0.0.0"
  },
  "websocket": {
    "port": 8008,
    "interface": "0.0.0.0"
  }
}
```

Then it can be specified on the command line like so:

```
helics_broker_server --web --zmq --config broker_server_config.json
```

The configuration will then make the REST web server accessible on any interface on port 8080 and a WebSocket server on port 8008. The port in use can be specified in a configuration file, or via command line such as

```
helics_broker_server --web --zmq --http_server_args="--http_port=80"
```

Arguments are passed to servers using an option in the form `--<server>_server_args`, and in that arg field `--<server>_port` and `--<server>_interface` are valid arguments. Valid server names are `http`, `websocket`, `zmq`, `'tcp'`, and `udp`, and eventually `mpi`. The http web server also acknowledges `HELICS_HTTP_PORT` as an environment variable. The websocket server acknowledges `HELICS_WEBSOCKET_PORT` for the port numbers of the respective servers.

REST API

The running webserver will start a process that can respond to HTTP requests.

HTTP actions

HTTP VERB	Description
GET	Make a query, usually with nothing in the message body
PUSH	most general command, usually for creating a broker, but other actions are possible
SEARCH	make a query mostly with data in the body
PUT	create a broker
DELETE	remove a broker

Parameters

parameter	Description
command	specify the command to use if not implied from the HTTP action, primarily PUSH
broker	The broker to target a focused request, or the name of a broker to create or delete
type	For commands that create a broker, this is the type of the broker to create
target	The actual object to target in a query
query	The query to execute on the specified target
args	The command line args to pass into a created broker

Valid commands for the `command` parameter in either JSON or the URI:

- `query`, `search` : run a query
- `create` : create a broker
- `delete`, `remove` : remove a broker

Websocket API

The websocket API will always respond in a JSON packet. For search/get operations where the response is a JSON value that JSON will be returned. for other responses, they are converted to a JSON.

For create/delete commands the response will be:

```
{
  "status": 0
}
```

```
{
  "status": 0,
  "value": "<query result>"
}
```

For queries that are not a json value the response will be:

```
{
  "status": 401, //or some other code
  "error": "error message"
}
```

For queries that did not result in a valid response the response will be:

```
{
  "status": 404,
  "error": "error message"
}
```

The status code corresponds to the most appropriate html error codes.

Making queries

As a demo case there is a `brokerServerTestCase` executable that can be built when compiling the main HELICS library from source. Running this example starts a webserver on the localhost using port 80.

The response to queries is a string either in plain text or json. For example:

```
localhost/brokers
```

will return

```
{
  "brokers": [
    {
      "address": "tcp://127.0.0.1:23408",
      "isConnected": true,
      "isOpen": false,
      "isRoot": true,
      "name": "brokerA"
    },
    {
      "address": "tcp://127.0.0.1:23410",
      "isConnected": true,
      "isOpen": true,
      "isRoot": true,
      "name": "brokerB"
    }
  ]
}
```

Other queries should be directed to a specific broker such as:

```
http://localhost/brokerA/brokers
```

which will produce a string vector:

```
[41888-wfQ8t-GIGjS-dndI3-e7zuk;41888-e9KF2-HAfm8-Rft0w-JLV4a]
```

The following:


```
http://localhost/brokerA/federate_map
```

will produce a map of the federates in the federation:

```
{
  "brokers": [],
  "cores": [
    {
      "federates": [
        {
          "id": 131072,
          "name": "fedA_1",
          "parent": 1879048192
        }
      ],
      "id": 1879048192,
      "name": "41888-wfQ8t-GIGjS-dndI3-e7zuk",
      "parent": 1
    },
    {
      "federates": [
        {
          "id": 131073,
          "name": "fedA_2",
          "parent": 1879048193
        }
      ],
      "id": 1879048193,
      "name": "41888-e9KF2-HAfm8-Rft0w-JLV4a",
      "parent": 1
    }
  ],
  "id": 1,
  "name": "brokerA"
}
```

Making an invalid query will produce:

```
http://localhost/brokerA/i_dont_care -> #invalid
```

Queries can be made in a number of different formats, the following are equivalent:

- `http://localhost/brokerA/publications`
- `http://localhost/brokerA`
- `http://localhost/brokerA?query=publications`
- `http://localhost/brokerA/publications`
- `http://localhost/brokerA/root/publications`
- `http://localhost?broker=brokerA&query=publications&target=root`

In the example, these will all produce `[pub1; fedA_1/pub_string]` which is a list of the publications. POST requests can also be made using a similar format.

Queries

Currently any query is accessible through this interface. Queries have a target and a query. The target is some named object in the federation and the query is a question. The available queries are listed [here](#). More are expected to be added.

Json

For both the websockets and REST API they can accept arguments in JSON format. For the REST API the parameters can be a combination of arguments in the URI and JSON in the body of the request For example:

```
{
  "command": "search",
  "broker": "broker1",
  "target": "federate0",
  "query": "current_state"
}
```

The most likely use case for this will be as a component for a more sophisticated control interface, so a more user friendly setup will be using the webserver as a back-end for control, debugging, information, and visualization of a running co-simulation.

Core Types

HELICS leverages a number of underlying communication technologies to achieve the coordination between the co-simulation components. Some of these technologies are designed to be general in nature (such as [ZeroMQ](#)) and others are designed for particular situations (such as [MPI](#) in HPC contexts).

There are several different core/broker types available in HELICS, each providing advantages in particular circumstances depending on the architecture of the federation and the underlying computation and network environment on which it is executing.

Generally speaking, the performance of the various cores is as follows (from best to worst)

1. MPI
2. IPC
3. UDP
4. TCP
5. ZMQ

Test

The Test core functions in a single process, and works through inter-thread communications. It's primary purpose is to test communication patterns and algorithms. However, in situations where all federates can be run in a single process it is probably the fastest and easiest to setup.

Interprocess (IPC)

The Interprocess core leverages Boost's interprocess communication (a part of the HELICS library) and uses memory-mapped files to transfer data rather than the network stack; in some circumstances it can be faster than the other cores. It can only be used inside a single, shared-memory compute environment (generally a single compute node). It also has some limitations on message sizes. It does not support multi-tiered brokers.

ZMQ

The ZMQ is the default core type and provides effective and robust communication for federations spread across multiple compute nodes. It uses the [ZMQ](#) mechanisms. Internally, it makes use of the REQ/REP mechanics for priority communications (such as [queries](#)) and PUSH/PULL for non-priority communication messages.

ZMQ_SS

The ZMQ_SS core also uses ZMQ for the underlying messaging technology but was developed to minimize the number of sockets in use, supporting very high federate counts on a single machine. It uses the DEALER/ROUTER mechanics instead of PUSH/PULL

UDP

UDP cores sends IP messages and carries with it the traditional limitation of UDP messaging: no guaranteed delivery or order of received messages. It may be faster in cases with highly reliable networking. Its primary use is for performance testing and the UDP core uses [asio](#) for networking.

TCP

TCP communications is an alternative to ZMQ on platforms where ZMQ is not available. Since the ZMQ messaging bus is built on-top of TCP it is expected that TCP provides higher performance than ZMQ. Performance comparisons have not been done, so it is unclear as to the relative performance differences between TCP, UDP, and ZMQ. It uses the [asio](#) library for networking

TCP_SS

The TCP_SS core uses TCP as the underlying messaging technology and is targeted at networking environments where it is convenient or required that outgoing connections be made from the cores or brokers but have only a single external socket exposed.

MPI

MPI communications is often used in HPC systems. It uses the message passing interface to communicate between nodes in an HPC system. It is still in testing and over time there is expected to be a few different levels of the MPI core used in different platforms depending on MPI versions available and federation needs.

Example

Generally, all federates in a federation utilize the same core type. There could be reasons for this not to be the case, though. For example, part of the federation could be running on MPI in an HPC environment while the rest is running on one or more compute nodes outside that environment. To allow the federates in the HPC environment to take advantage of the high-speed MPI bus but to allow the rest of the federation without access to MPI to use ZMQ, a “multi-broker” or “multi-protocol broker” must be set up.

A full co-simulation example showing how to implement a multi-core federation *is written up [here](#)* (and the source code can be found [HELICS Examples repository](#)).

Multi-Protocol Broker

Starting in HELICS 2.5 there is a Multi Broker type that allows connection with multiple communication types simultaneously. The multibroker allows an unlimited number of communication operations to interact.

Starting a multiBroker

A multibroker can be started as BrokerApp or a helics_broker. For the HELICS Broker the configuration must be given as a file since each of the core types linked must be configured independently. Using the helics_broker the startup commands would look something like

```
helics_broker --type multi --config=helics_mb_config.json --name=broker1
```

A couple example configurations follow.

```
{
  "master": {
    "type": "test"
  },
  "comms": [
    {
      "type": "zmq",
      "interfaceport": 23410
    },
    {
      "type": "zmq",
      "interfaceport": 23700
    }
  ]
}
```

The primary communication pathway can be specified in a master object or on the root of the configuration file.

```
{
  "type": "test",
  "comms": [
    {
      "type": "zmq"
    },
    {
      "type": "tcp"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    }
  ]
}
```

Master comm information can also be given through the command line. The master comm is the only one in which higher level broker information may be specified. Any broker related specification in the comms sections will result in an error. If the MultiBroker is intended to be a root broker then no master section is required. Multiple network communication pathways of the same type are allowed assuming they use different ports.

Programmatically multibrokers can also be started using the BrokerApp and giving it the type `helics::core_type::MULTI` for arguments to the multibroker the type of the master comm can be specified on the command line arguments as well.

Limitations

- Using TCPSS comms in the multibroker does not currently support outgoing connections like a full TCPSS broker would. This will likely be fixed in upcoming releases.
- Configuration files must currently be in JSON, in a few limited cases TOML files may work, but configuration of multiple comms in a toml file will not work. This will also likely be fixed in upcoming releases.
- General support for multibrokers is not provided in the webServer due to limitations on the configuration files. Some mechanism for this will be allowed in a future release.

Example

An example implementation of a multi-protocol broker with explanation [can be found here](#) with the [source code over here in the repository](#).

Broker Hierarchies

The simplest and most straight-forward way HELICS co-simulations are constructed is with a single broker. If all federates are running on a single compute node, a single broker is likely all you'll need. In situations where the co-simulation is running across multiple compute nodes, with a little bit of planning, the use of a hierarchy of brokers can help improve co-simulation performance.

Why Multiple Brokers?

Brokers are primarily concerned with facilitating the message exchanges between federates. As the number of messages that must be passed increase, the load on the broker(s) increase as well. In cases where the co-simulation is deployed across multiple compute nodes, the total cost of sending messages over the network (primarily in terms of latency and the corresponding slowdown in federation performance) can become non-trivial. The worst case develops when the compute node hosting the broker is physically distant (and thus, experiences higher latency) from the compute nodes where the individual federates are running. If two federates running on the same node need to talk to each other but have a high-latency connection to the broker, their communication will be significantly hampered as they coordinate with that distant broker.

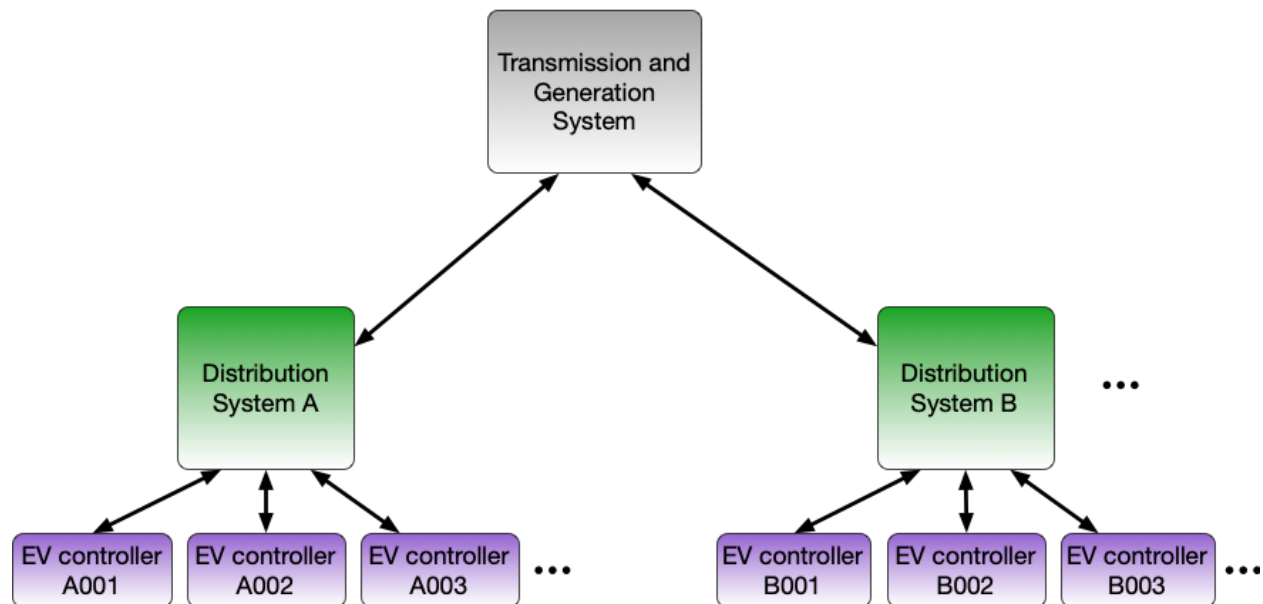
And thus the solution presents itself: a broker local to the compute node where the federates are located will have a very low latency connection and will generally experience a lower load of messages it needs to process. To receive this benefit, though, the co-simulation needs to be deployed to the compute nodes in such a way that the federates that talk to each other most frequently/heavily are located on the same node and a broker is also deployed to that node.

When federates join the federation they can be assigned to specific brokers and when the co-simulation begins, each local broker uses this information to route the messages it can. Any messages that it is not connected to, it sends up the broker hierarchy. The broker at the top of the hierarchy is the broker of last resort and it is guaranteed to be able to route all messages down the hierarchy to their intended destination.

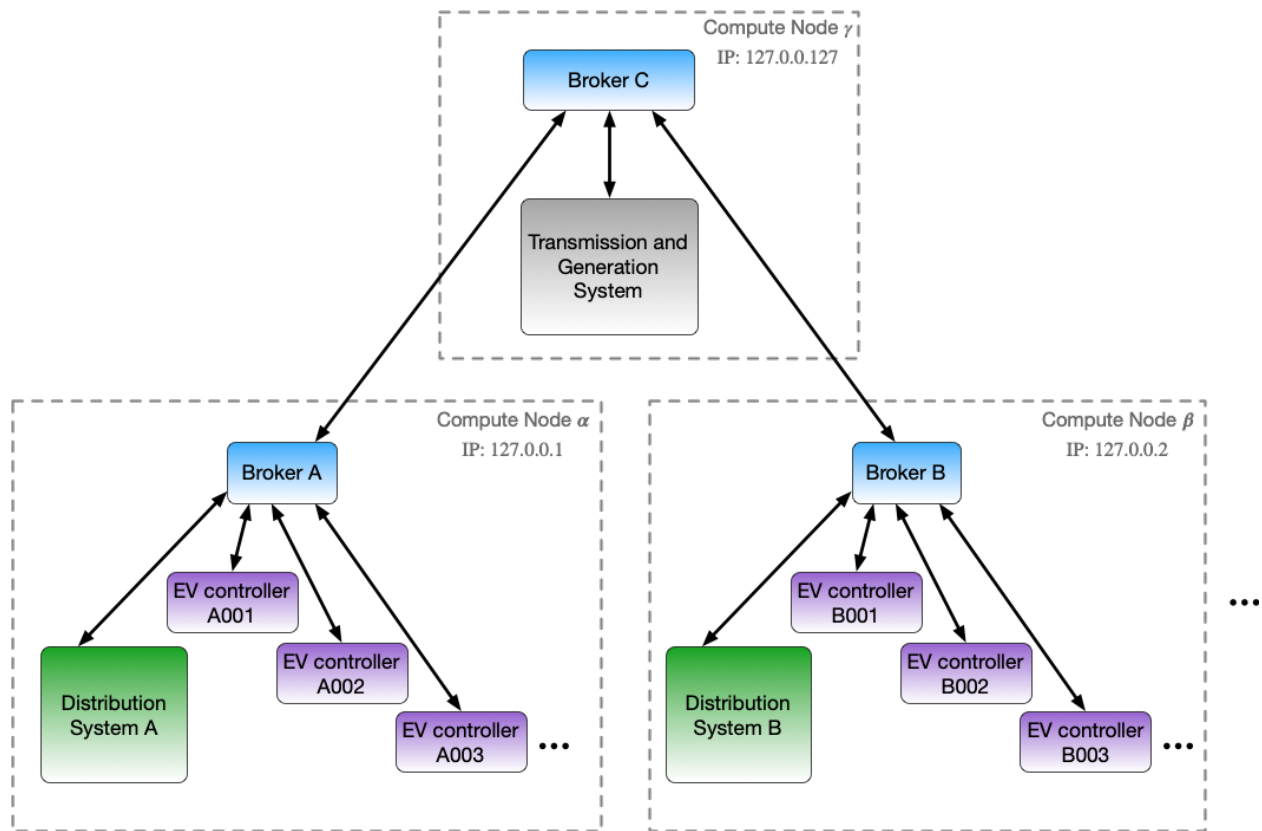
So, using an example we've seen several times, imagine a scenario where a single transmission system covering the western US is being simulated with many, many individual neighborhood level power systems attached to that regional system and controllers manage a hypothetical fleet of electric vehicles (EVs) whose owners are in these neighborhoods. The EV controllers and the distribution system federates interact frequently and the distribution system federates and the transmission system federates also interact frequently.

The diagrams below show the message and broker topologies for this hypothetical examples.

Message Topology



Broker Topology



Broker Hierarchy Implementation

To implement a broker hierarchy, modifications to the configuration files for the federates and command line options for the brokers need to be made.

For examples, the config JSON for the Distribution System A (where Broker A is at IP 127.0.0.1) would look something like this:

```
{
  "name" : "DistributionSystemA",
  "coreInit": "--broker_address=tcp://127.0.0.1"
  ...
}
```

The command line for launching Broker A also needs to be adjusted. For this examples, let's assume there is a total of 200 federates are expected by Broker A. Note that `broker_address` is used to define the address of the broker next up in the hierarchy; in this, case Broker C.

```
helics_broker -f200 --broker_address=tcp://127.0.0.127
```

Lastly, the JSON config file for the Transmission and Generation System federate needs to indicate where it's broker (Broker C) is at (IP 127.0.0.127):

```
{
  "name" : "TransmissionGenerationSystem",
  "coreInit": "--broker_address=tcp://127.0.0.127"
  ...
}
```

Example

A full co-simulation example showing how to implement a broker hierarchy *has been written up over here* (and the source code is in the [HELICS Examples repository](#)).

Environment variables

The HELICS command line processor has some ability to read and interpret command line environment variables. These can assist in setting up co-simulations. In general the configuration of HELICS comes from 3 sources during setup. After setup API's exist for changing the configuration later. The highest priority is given to command line arguments. The second priority is given to configuration files, which can be given through a command string such as `--config=configFile.ini`. The file can be a `.ini`, `.toml`, or `.json`. By default HELICS looks for a `helicsConfig.ini` file. The lowest priority option is though environment variables. Only a subset of controls work with environment variables. All environment variables used by HELICS begin with `HELICS_`.

Federate environment variables

For setting up a federate a few environment variables are used

- `HELICS_LOG_LEVEL`: the log level for the federate to use. Equivalent to `--loglevel=X`
- `HELICS_CORE_INIT_STRING`: the init string to pass to the core when creating it. Equivalent to `--coreinit=X`
- `HELICS_CORE_TYPE`: the type of core to use e.g. "ZMQ", "TCP", "IPC", "MPI", etc. Equivalent to specifying `--coretype=X`

Core and Broker environment variables

- `HELICS_BROKER_LOG_LEVEL`: the log level for the broker to use. Equivalent to `--loglevel=X`
- `HELICS_BROKER_KEY`: the key to use for connecting a core to a broker. See broker key
- `HELICS_BROKER_ADDRESS`: the interface address of the broker. Equivalent to `--brokeraddress=X`
- `HELICS_BROKER_PORT`: the port number of the broker. Equivalent to `--brokerport=X`
- `HELICS_CONNECTION_PORT`: the port number to use for connecting. This has different behavior for cores and brokers. For cores this is the broker port and for brokers this is the local port
- `HELICS_CONNECTION_ADDRESS`: the interface address to use for connecting. This has different behavior for cores and brokers. For cores this is the broker address and for brokers this is the interface.
- `HELICS_LOCAL_PORT`: the port number to use on the local interface for external connections. Equivalent to `--localport=X`
- `HELICS_BROKER_INIT`: the command line arguments to give to an autogenerated broker. Equivalent to `--brokerinit=X`

- `HELICS_CORE_TYPE` : the type of core to use e.g. “ZMQ”, “TCP”, “IPC”, “MPI”, etc. Equivalent to specifying `--coretype=X`

Simultaneous Co-simulations

Sometimes it is necessary or desirable to be able to execute multiple simultaneous simulations on a single computer system. Either for increased parallelism or from multiple users or as part of a larger coordinated execution for sensitivity analysis or uncertainty quantification. HELICS includes a number of different options for managing this and making it easier.

General Notes

HELICS starts with some default port numbers for network communication, so only a single broker (per core type) with default options is allowed to be running on a single computer at a given time. This is the general restriction on running multiple simultaneous co-simulations. It is not allowed to have multiple default brokers running at the same time, the network ports will interfere and the co-simulation will fail.

There are a number of ways around this and some tools to assist in checking and coordinating.

Specify port numbers

The manual approach works fine. All the network core types accept user specified port numbers. The following script will start up two brokers on separate port numbers:

```
helics_broker --type=zmq --port=20200 &
helics_broker --type=zmq --port=20400 &
```

Federates connecting to the broker would need to specify the `--brokerport=X` to connect with the appropriate broker. These brokers operate independently of each other. The port numbers assigned to the cores and federates can also be user assigned but if left to default will be automatically assigned by the broker and should not interfere with each other.

An example of configuring multiple federations to run on a single compute node using port numbers *has been written up over [here](#)* (and the source code can be found in the [HELICS Examples repo](#)).

Use Broker server

For the zmq, zmqss, tcp, and udp core types it is possible to use the broker server.

```
helics_broker_server --zmq
helics_broker_server --zmqss
helics_broker_server --tcp
helics_broker_server --udp
```

multiple broker servers can be run simultaneously

```
helics_broker_server --zmq --tcp --udp
```

The broker server currently has a default timeout of 30 minutes on the default port and will automatically generate brokers on separate ports and direct federates which broker to use. The duration of the server can be controlled via

```
helics_broker_server --zmq --duration=24hours
```

It will also generate brokers as needed so the `helics_broker` does not need to be restarted for every run.

By default the servers will use the default ports and all interfaces. This can be configured through a configuration file

```
helics_broker_server --zmq --duration=24hours --config=broker_config.json
```

this is a json file. The sections in the json file include the server type For example

```
{
  "zmq": {
    "interface": "tcp://127.0.0.1"
  },
  "tcp": {
    "interface": "127.0.0.1",
    "port": 9568
  }
}
```

There is also a *webserver* that can be run with the other broker servers.

Use of keys

If there are multiple users and you want to verify that a specific broker can only be used with federates you control. It is possible to add a key to the broker that is required to be supplied with the federates to connect to the broker. **NOTE:** *this is not a cryptographic key, it is just a string that is not programmatically accessible to others.*

```
helics_broker --type=zmq --key=my_broker_key
```

Federates then need to supply the key as part of the configuration string otherwise the broker will return an error on connection. This is like a fence that prevents some accidental interactions. The rule is that both the federate and broker must provide no key or the same key.

Orchestration for HPC systems

The goal of this guide is to show and guide you on how to handle co-simulation orchestration on high-performance computing systems. We will walk through using a specific tool (*Merlin*) that has been tested with HELICS co-simulations. This is not the only tool that exists that has this capability and is not a requirement for co-simulation orchestration. One advantage that Merlin has is its ability to interface with HPC systems that have *SLURM* or *Flux* as their resource managers.

Definition of “Orchestration” in HELICS

We will define the term “orchestration” within HELICS as workflow and deployment in an HPC environment. This will allow users to define a co-simulation workflow they would like to execute and deploy the co-simulation either on their own machine or in an HPC environment.

Orchestration with Merlin

First you will need to build and install [Merlin](#). This guide will walk through a suggested co-simulation spec using Merlin to launch a HELICS co-simulation. This is not a comprehensive guide on how to use Merlin, but a guide to use Merlin for HELICS co-simulation orchestration. For a full guide on how to use Merlin, please refer to the [Merlin tutorial](#).

Merlin is a distributed task queuing system, designed to allow complex HPC workflows to scale to large numbers of simulations. It is designed to make building, running, and processing large scale HPC workflows manageable. It is not limited to HPC; it can also be set up on a single machine.

Merlin translates a command-line focused workflow into discrete tasks that it will queue up and launch. This workflow is called a specification, spec for short. The spec is separated into multiple sections that is used to describe how to execute the workflow. This workflow is then represented as a directed acyclic graph (DAG) which describes how the workflow executes.

Once the Merlin spec has been created, the main execution logic is contained in the Study step. This step describes how the applications or scripts need to be executed in the command line in order to execute your workflow. The study step is made up of multiple run steps that are represented as the nodes in the DAG.

For a more in-depth explanation on how Merlin works, take a look at their documentation [here](#)

Why Merlin

The biggest feature that Merlin will give HELICS users is its ability to deploy co-simulations in an HPC environment. Merlin has the ability to interface with both FLUX and SLURM workload managers that are installed on HPC machines. Merlin will handle the request for resource allocation, and take care of job and task distribution amongst the nodes. Users will not need to know how to use SLURM or FLUX because Merlin will handle all resource allocation calls to the workload manager, the user will only need to provide the number of nodes they need for their study.

Another benefit of using Merlin for HELICS co-simulation is its flexibility to manage complex co-simulations. Another tool that you may have heard of is `helics_cli`. `helics_cli` is a command-line tool to launch HELICS co-simulations. It currently as of this writing does not have the ability to analyze the data and launch subsequent co-simulations. In this type of scenario a user could use Merlin to setup a specification that included an analyze step in the Study step of Merlin. The analysis step would determine if another co-simulation was needed and the input to the next co-simulation, and would then proceed to launch the co-simulation with the input generated by the analysis step.

Merlin Specification

A Merlin specification has multiple parts that control how a co-simulation may run. Below we describe how each part can be used in a HELICS co-simulation workflow. For the sake of simplicity we are using the the pi-exchange python example that can be found [here](#). The goal will be to have Merlin launch multiple pi-senders and pi-receivers.

Merlin workflow description and environment

Merlin has a description and an environment block. The `description` block provides the name and a short description of the study.

```
description:
  name: Test helics
  description: Juggle helics data
```

The `env` block describes the environment that the study will execute in. This is a place where you can set environment variables to control the number of federates you may need in your co-simulation. In this example, `N_SAMPLES` will be used to describe how many pi-senders and pi-receivers (total federates) we want in our co-simulation.

```
env:
  variables:
    OUTPUT_PATH: ./helics_juggle_output
    N_SAMPLES: 8
```

Merlin Step

The Merlin step is the input data generation step. This step describes how to create the initial inputs for the co-simulation so that subsequent steps can use this input to start the co-simulation. Below is how we might describe the Merlin step for our pi-exchange study.

```
merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/make_samples.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/pireceiver.py $(MERLIN_INFO)
        cp $(SPECROOT)/pisender.py $(MERLIN_INFO)
      file: samples.csv
      column_labels: [FED]
```

NOTE: `samples.csv` is generated by `make_samples.py`. Each line in `samples.csv` is a name of one of the json files that is created.

There is a python script called `make_samples.py` located in the [HELICS repository](#) that generates all helics-cli json configs that will be executed by helics-cli that will be used to execute the co-simulations. `N_SAMPLES` is an environment variable that is set to 8, so in this example 8 pireceivers and 8 pisenders will be created and used in this co-simulations. `make_samples.py` also outputs the name of each json file to a csv file called `samples.csv`. `samples.csv` contains the names of the json files that were generated. The `column_labels` tag tells Merlin to set each column in `samples.csv` to `[FED]`. This means we can use `FED` as a variable in the study step. Below is an example of one of the json files that is created.

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "python3 -u pisender.py 0",
      "host": "localhost",
      "name": "pisender0"
    }
  ],
  "name": "pisender0"
}
```

This json file will then be used as the input file for helics-cli. The helics-cli will be executed in the study step in Merlin which we will go over next.

Study Step

The study step is where Merlin will execute all the steps specified in the block. Each step is denoted by a name and has a run segment. The run segment is where you will tell Merlin what commands need to be executed.

```
- name: start_federates <-- Name of the step
description: say Hello
run:
  cmd: |
    helics run --path=$(FED) <-- execute the helics_cli for each column in samples.csv
    echo "DONE"
```

In the example snippet we ask Merlin to execute the json file that was created in the Merlin step. Since the FED variable is a list, this command will get executed for each index in FED.

Full Spec

Below is the full Merlin spec that was created to make 8 pi-receivers and pi-senders and execute it as a Merlin workflow.

```
description:
  name: Test helics
  description: Juggle helics data

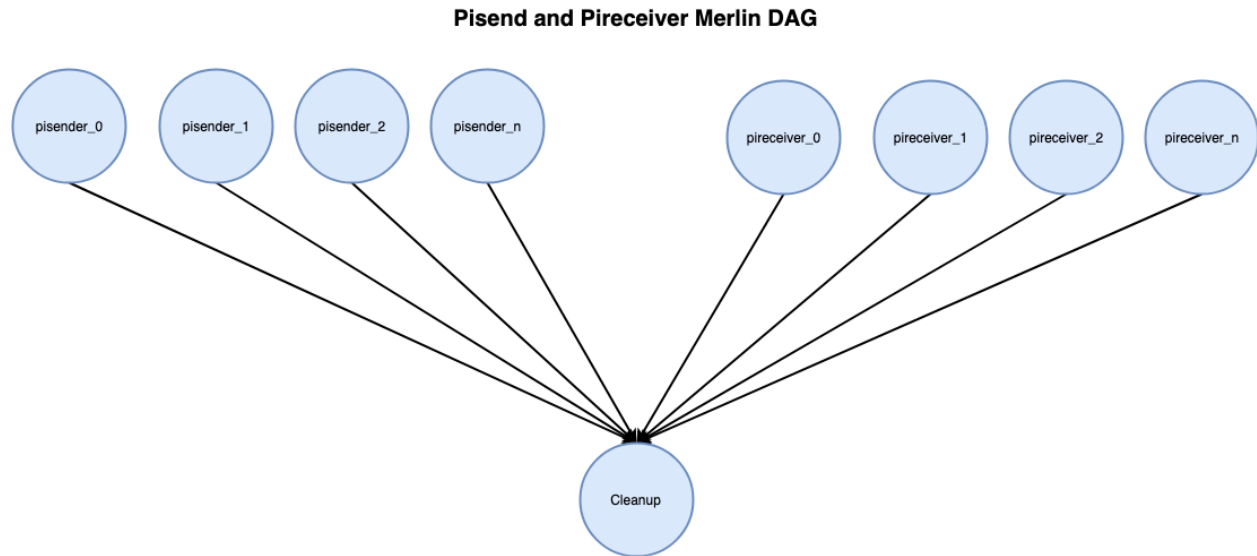
env:
  variables:
    OUTPUT_PATH: ./helics_juggle_output
    N_SAMPLES: 8

merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/make_samples.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/pireceiver.py $(MERLIN_INFO)
        cp $(SPECROOT)/pisender.py $(MERLIN_INFO)
      file: samples.csv
      column_labels: [FED]

study:
  - name: start_federates
    description: say Hello
    run:
      cmd: |
        spack load helics
        helics run --path=$(FED)
        echo "DONE"
  - name: cleanup
    description: Clean up
    run:
      cmd: rm $(SPECROOT)/samples.csv
      depends: [start_federates_*]
```

DAG of the spec

Finally, we can look at the DAG of the spec to visualize the steps in the Study.



Orchestration Example

An example of orchestrating multiple simulation runs (e.g. Monte Carlo co-simulation) is given in the [Advanced Examples](#) Section.

Terminating HELICS

If executing from a C or C++ based program. Ctrl-C should do the right thing, and terminate the local program. If the co-simulation is running across multiple machines then the remaining programs won't terminate properly and will either timeout or if that was disabled potentially deadlock.

Signal handler facilities

The C shared library has some facilities to enable a signal handler.

```
/** load a signal handler that handles Ctrl-C (SIGINT) and shuts down the library*/  
void helicsLoadSignalHandler();  
  
/** clear HELICS based signal Handlers*/  
void helicsClearSignalHandler();
```

This function will insert a signal handler that generates a global error on known objects and waits a certain amount time, clears the print buffer, and terminates.

NOTE : the signal handlers use unsafe operations, so there is no guarantee they will work, or that they will work as expected. Testing indicates they work in most situations and improve operations where needed but it is not 100% reliable or safe code. They make use of atomic variables, mutexes, and other constructs that are not technically safe in signal handlers. The primary use case is program termination so the effects are minimized and they usually work, but the unsafe nature of them should be kept in mind.

```

/** load a custom signal handler to execute prior to the abort signal handler
@details This function is not 100% reliable it will most likely work but uses some
↳ functions and
techniques that are not 100% guaranteed to work in a signal handler
and in worst case it could deadlock. That is somewhat unlikely given usage patterns
but it is possible*/
void helicsLoadSignalHandlerCallback(helics_bool (*handler)(int));

```

It is also possible to insert a custom callback into the signal handler chain. Again this is not 100% reliable. But is useful for some language API's that do other things to signals. This allows for inserting a custom operation that does some other cleanup. The callback has a helics_boolean return value. If the value is set to helics_true(or any positive value) then the normal Signal handler is called which aborts ongoing federations and exits. If it is set to helics_false then the default callback is not executed.

Signal handlers in C++

Facilities for signal handling in C++ were not put in place since it is easy enough for a user to place their own handlers which would likely do a better job than any built in ones, so a default one was not put in place at present though may be at a later time.

Generating an error

A global error generated anywhere in a federation will terminate the co-simulation.

```

/**
 * generate a global error through a broker this will terminate the federation
 *
 * @param broker The broker to set the time barrier for.
 * @param errorCode the error code to associate with the global error
 * @param errorString an error message to associate with the error
 * @param[in,out] err An error object that will contain an error code and string if any.
↳ error occurred during the execution of the function.
 */
void helicsBrokerGlobalError(helics_broker broker, int errorCode, const char_
↳ *errorString, helics_error* err);

void helicsCoreGlobalError(helics_core core, int errorCode, const char* errorString,
↳ helics_error* err);

/**
 * Generate a global error from a federate.
 *
 * @details A global error halts the co-simulation completely.
 *
 * @param fed The federate to create an error in.
 * @param error_code The integer code for the error.
 * @param error_string A string describing the error.
 */
HELICS_EXPORT void helicsFederateGlobalError(helics_federate fed, int error_code, const_
↳ char* error_string);

```

(continues on next page)

Corresponding functions are available in the C++ API as well. Any global error will cause a termination of the co-simulation.

Some modifying flags

Setting the `helics_terminate_on_error` flag to true will escalate any local error into a global one and terminate the co-simulation. This includes any mismatched configuration or other local issues.

Comments

Generally it isn't a wise idea to just terminate the co-simulation without letting everyone else know. If you control everything it probably works fine but as co-simulations get larger more care needs to be taken to prevent zombie processes and hung federates and brokers. Which can cause issues on the next one. This is an evolving area of how best to handle terminating large co-simulations in abnormal conditions and hopefully the best practices will make it easier for users.

Profiling

As of versions 2.8 or 3.0.1 HELICS includes a basic profiling capability. This is simply the capability to generate timestamps when entering or exiting HELICS blocking call loops where a federate may be waiting on other federates.

Output

The profiling output can be either in the other log files or a separate file, and can be enabled at the federate, core, or broker levels. There are 3 messages which may be observed:

```
<PROFILING>test1[131072](created)MARKER<138286445040200|1627493672761320800>[t=-
↪9223372036.854776]</PROFILING>
<PROFILING>test1[131072](initializing)HELICS CODE ENTRY<138286445185500>[t=-10000000]</
↪PROFILING>
<PROFILING>test1[131072](executing)HELICS CODE EXIT<138286445241300>[t=0]</PROFILING>
<PROFILING>test1[131072](executing)HELICS CODE ENTRY<138286445272500>[t=0]</PROFILING>
```

The messages all start and end with and to make an xml-like tag. The message format is `FederateName[FederateID](federateState)MESSAGE<wall-clock time>[simulation time]`

The federate state is one of `created`, `initializing`, `executing`, `terminating`, `terminated`, or `error`.

The three possible MESSAGE values are:

- **MARKER** : A time stamp matching the local system up time value with a global time timestamp.
- **HELICS CODE ENTRY** : Indicator that the executing code is entering a HELICS controlled loop
- **HELICS CODE EXIT** : Indicator that the executing code is returning control back to the federate.

For **HELICS CODE ENTRY** and **HELICS CODE EXIT** messages the time is a steady clock time, usually the time the system on which the federate is running has been up. The **MARKER** messages have two timestamps for global coordination, `<steady clock time|system time>`. The system time is the wall clock time as available by the system, which is usually with reference to Jan 1, 1970 and in GMT.

The timestamp values are an integer count of nanoseconds. For all 3 message types they refer to the system uptime which is monotonically non-decreasing and steady. This value will differ from each computer on which federates are running, though. To calibrate for this there is a marker that gets triggered when the profiling is activated, indicating the local uptime that is synchronous across compute nodes. This matches a system uptime, with the global system time. The ability to match these across multiple machines will depend on the latency associated with time synchronization across the utilized compute nodes. No effort is made in HELICS to remove this latency or even measure it; that is, though the marker time is measured in nanoseconds it could easily differ by microseconds or even milliseconds depending on the networking conditions between the compute nodes.

Enabling profiling

Profiling can be enabled at any level of the hierarchy in HELICS and when enabled it will automatically enable profiling on all the children of that object. For example, if profiling is enabled on a broker, all associated cores will enable profiling and all federates associated with those cores will also have profiling enabled. This propagation will also apply to any child brokers and their associated cores and federates.

Broker profiling

Profiling is enabled via the command prompt by passing the `--profiler` option when calling `helics_broker`.

- `--profiler=save_profile2.txt` will save profiling data to a text file `save_profile2.txt`
- `--profiler=log` will capture the profile text output to the normal log file or callback
- `--profiler` is the same as `--profiler=log`

Enabling this flag will pass in the appropriate flags to all children brokers and cores.

Core profiling

Profiling is enabled via the `coreinitstring` by adding a `--profiler` option.

- `--profiler=save_profile2.txt` will save profiling data to a text file `save_profile2.txt`
- `--profiler=log` will capture the profile text output to the normal log file or callback
- `--profiler` is the same as `--profiler=log`

Enabling this flag will pass in the appropriate flags to all children federates.

Federate profiling

Profiling on a federate will recognize the same flags as a core, and pass them as appropriate to the core. However a federate also supports passing the flags and a few additional ones into the federate itself.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_PROFILING, HELICS_TRUE, &err);
```

can directly enable the profiling. If nothing else is set this will end up generating a message in the log of the root broker.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_PROFILING_MARKER, HELICS_TRUE, &err);
```

can generate an additional marker message if logging is enabled.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_LOCAL_PROFILING_CAPTURE, HELICS_TRUE, &err);
```

captures the profiling messages for the federate in the federate log instead of forwarding them to the core or broker.

Some can be set through the flags option for federate configuration. `--flags=profiling`, `local_profiling_capture` can be set through command line or configuration files. If enabling the `local_profiling_capture`, profiling must also be enabled; that is, just setting `local_profiling_capture` does not enable profiling. The profiling marker doesn't make sense anywhere but through the program call.

Notes

This capability is preliminary and subject to change based on initial feedback. In HELICS 3 there will probably be some additional command infrastructure to handle profiling as well added in the future.

If timing is done in the federate itself as well there will be a time gap; there is some processing code between the profiling message, and when the actual function call returns, but it is not blocking and should be fairly short, though dependent on how much data is actually transferred in the federate.

Targeted Endpoints

Endpoints can be configured to send messages to other specific endpoints by two means, `default` and `targeted`.

- **default** - Convenient way of specifying the intended destination of a message from the configuration file. Allows the API call that sends the message to leave the destination field blank. DOES NOT create a specific dependency between the sending and receiving endpoint for the HELICS core library to use when figuring out which federate should be granted what time.
- **targeted** - Creates a dependency link between the sending and receiving endpoints, just like in value exchanges. This allows the HELICS core library to make more efficient decisions about when to grant times to federates and can lead to more efficient co-simulation execution. Targeted endpoints can specify a list of federates that all their messages go to.

Targeted endpoint configuration example:

```
{
  "name": "EV_Controller",
  "coreType": "zmq",
  "timeDelta": 1.0,
  "endpoints": [
    {
      "name": "EV_Controller/EV6",
      "global": true,
      "destinationTarget": "charger/ep"
    }
  ]
}
```

Dynamic Federations

In general a dynamic federation is one in which the federates or simulators are not all available at the start of the co-simulation. Instead some components attach to the running co-simulation part-way through the simulation. The longer term goal of HELICS is to fully support dynamic federations but there are many complicated issues to handle timing and data communication that need to be resolved before full general dynamic federations are supported. As of HELICS 3.1 a limited though quite useful form is available through observer federates.

Levels of Dynamic Federations

Dynamic Federations come in various flavors of complexity. The first is simply allowing additional subscriptions to existing federates. The next is allowing additional “observer” federates to join a running co-simulation. The third is allowing new publications, endpoints, or filters on existing federates. And the fourth is a full dynamic federation.

The first of these has been unofficially available for some time and will be supported in HELICS 3.1. Dynamic observer federates will also be allowed in HELICS 3.1, and will be detailed further in a subsequent section. The last two levels are currently a work in progress and are currently planned to be supported in HELICS 3.2 (Early 2022).

Dynamic Subscriptions

In the normal case, the `registerInput`, `registerSubscription`, or `registerTargetedEndpoint` are done in the creation phase of co-simulation so all the checks and connections can be made and everything set up before `timeZero` (start of co-simulation). If these calls are made after the `enterInitializingMode` call, then the time guarantees are different. Ordinarily data published in the initialization phase would be available to all subscriptions of a publication, but with dynamic subscriptions the data is not going to be sent to the new subscription until new publications after a time request or `enterExecutingMode` call has been made. (A `bufferData` flag is in the works to allow data to be sent for dynamic subscriptions at the time of connection if set on the publisher).

Apart from time considerations there are no additional requirements. The checks on the subscription are done immediately as opposed to waiting for the `enterInitializingMode` call.

Dynamic Observer Federates

A federate may declare itself to be an observer in the `FederateInfo` structure when a federate is declared. This can be done via the command line (`--observer`) or through a flag.

In the C++ API

```
FederateInfo fi;
fi.observer=true;
```

and in the C or other language API's

```
helicsFederateInfoSetFlagOption(fi, HELICS_FLAG_OBSERVER, HELICS_TRUE, &err);
```

The observer flag triggers some flags in the created core and federate to notify the broker that it can be dynamically added. Otherwise it will get an error notification that the federate is not accepting new federates. If a core is created before the new federate it must also be created with the observer flag enabled. Otherwise, it will not be allowed to join the federation.

Once joined, subscriptions can be added with the same timing rules as described in the previous section. One key thing to be aware of is that for dynamic federates the time returned after `enterExecutingMode` is not necessarily 0, but will

depend on the time of federates containing the publications or endpoints that are being linked. If there are no data pathways zero will still be returned.

The utility of this capability is primarily for debugging/observation purposes. It is possible to join, get the latest data and make some queries about the current co-simulation status for monitoring purposes and disconnect. At present a new name is required each time an observer connects. This may be relaxed in the future.

Dynamic Publications

Dynamic publications and endpoints will be allowed in the near future. These would include just creating a publication or endpoint on an already executing federate. It would then be available for others to connect to and receive data. It would typically take at least two timesteps of the publisher to actually receive any data from the new publication. The first to ensure registration is completed and the second to publish the data. It would be possible to send the data immediately after the register call but it would be impossible (unless on the same core) for any other federate to connect until after the second request time call.

Full Dynamic Federations

Allowing full dynamic federations will require activating a flag on the root broker to explicitly allow dynamic federations. It is not intended to be a normal use case, and will not be enabled by default. This functionality is not operational as of HELICS 3.1 and significantly more testing is needed before the feature becomes generally available and more details will be added at that time.

Whereas the *Fundamental Topics* provided a broad overview of co-simulation and a good step-by-step introduction to setting up a HELICS co-simulation, the Advanced Topics section assumes you, the reader, have a familiarity and experience with running HELICS co-simulations. If that's not the case, it's well worth your while to go review the *Fundamental Topics* and *corresponding examples*. In this section it will be assumed you know things like:

- The difference between value and message passing in HELICS
- How to configure HELICS federate appropriately
- Familiarity with the common HELICS APIs (*e.g.* requesting time, getting subscribed values, publishing values)
- Experience running HELICS co-simulations

The Advanced Topics section will dig into specific features of HELICS that are less commonly used but can be very useful in particular situations. Each section below provides a description of the feature, what it does, the kind of use case that might utilize it, and then links to examples that demonstrate an implementation. It's important to note that there are many other HELICS features and APIs not demonstrated here that can also be useful. As they say in academia, we'll leave it as an exercise to the reader to discover these. (Hint: The *API references* and the *Configuration Options Reference* are good starting points to see what's out there in the broader HELICS world.)

The Advanced Topics will cover:

- **Multi-Source Inputs** - Using inputs (rather than subscriptions), it is possible to accept value signals from multiple sources. This section discusses the various tools HELICS provides for managing how to handle/resolve what can be conflicting or inconsistent signal data.
- **Architectures** - Introduction to different ways to connect federates, cores, and brokers to manage efficient passing of signals in a co-simulation.
- **Queries** - How queries can be used to get information on HELICS brokers, federates, and cores.
- **API Webserver** - How to interact with a running co-simulation using queries.
- **Cores** - Discussion of the different types of message-passing buses and their implementation as HELICS cores.
- **Multiple Brokers**

- *Connecting Multiple Core Types (Multi-Protocol Broker)* - What to do when one type of communication isn't sufficient.
- *Broker Hierarchies* - Purpose of broker hierarchies and how to configure a HELICS co-simulation to implement one.
- *Environment Variables* - HELICS supports some environment variables for configuration of a federate or broker.
- *Simultaneous co-simulations* - Options for running multiple independent co-simulations on a single system.
- *Orchestration Tool (Merlin)* - Brief guide on using [Merlin](#) to handle situations where a HELICS co-simulation is just one step in an automated analysis process (*e.g.* uncertainty quantification) or where assistance is needed deploying a large co-simulation in an HPC environment.
- *Program termination* - Some additional features in HELICS related to program shutdown and co-simulation termination.
- *Profiling* - Some profiling capability for co-simulations.
- *Targeted Endpoints* - details on the new targeted endpoints in HELICS 3.
- *Dynamic Federations* - Sometimes it is useful to have a federate that is not ready at the beginning of co-simulation. This is a dynamic federation. There are various levels of this (not all are available yet) and this document discusses some aspects of dynamic co-simulation.

1.4 Examples

The examples provided in the user guide begin with a simple co-simulation that you should be able to execute with only python and HELICS installed. If you have not installed HELICS yet, navigate to the [installation page](#).

What are we modeling?

The model for the examples is a co-simulation of a number of electric vehicle (EV) batteries and a charging port. A researcher may pose the question, “What is the state of charge of batteries on board EVs as they charge connected to a charging port?”

This can be addressed with a simple two-federate co-simulation, as demonstrated in the Fundamental Examples, or with a more complicated multi-federate co-simulation modeled in the Advanced Examples. In each learning path, modules are provided to the user to demonstrate a skill. The Advanced Examples build on the basics to make the co-simulation better emulate reality.

Learning Tracks

There are two learning tracks available to those hoping to improve their HELICS skills. The Fundamental Examples are designed for users with no experience with HELICS or co-simulation. The Advanced Examples are geared towards users who are familiar with HELICS and feel confident in their abilities to build a simple co-simulation. The Advanced Examples harness the full suite of HELICS capabilities, whereas the Fundamental Examples teach the user the basics.

These two learning tracks each start with a “base” model, which should also be considered the recommended default settings. Examples beyond the base model within a track are modular, not sequential, allowing the user to self-guide towards concepts in which they want to gain skill.

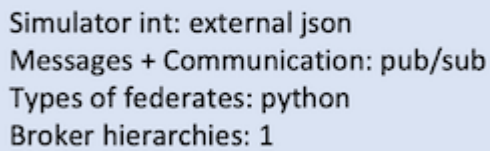
1.4.1 Fundamental Examples

The Fundamental Examples teach three concepts to build on a default setup:

Base Example Co-Simulation

The Base Example walks through a simple HELICS co-simulation between two python federates. This example also serves as the recommended defaults for setting up a co-simulation.

Default HELICS setup:



Simulator int: external json
Messages + Communication: pub/sub
Types of federates: python
Broker hierarchies: 1

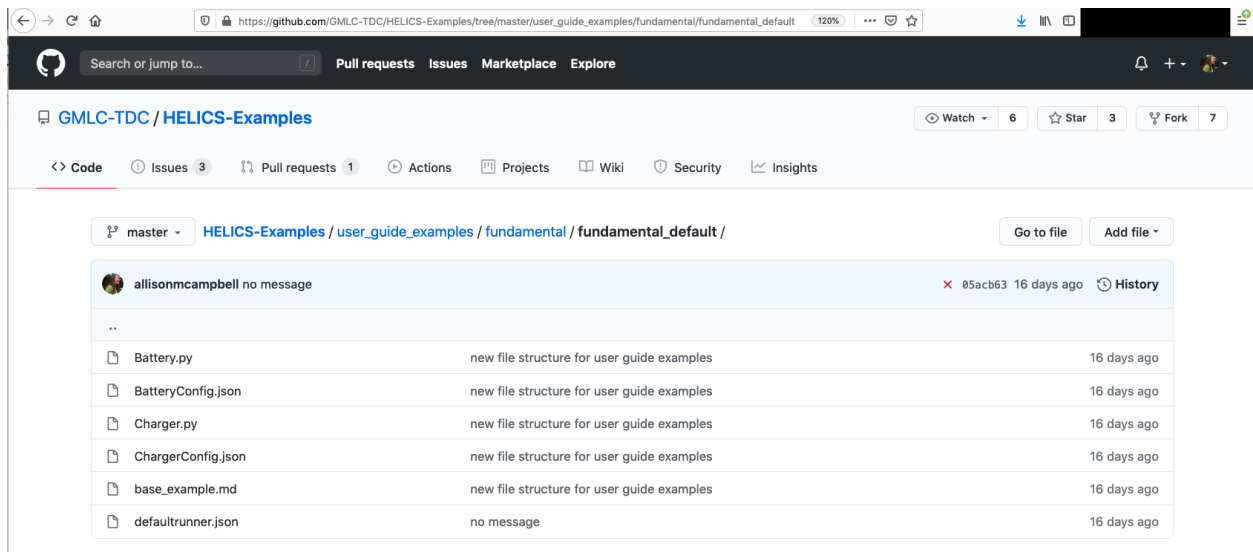
The base example described here will go into detail about the necessary components of a HELICS program. Subsequent examples in the Fundamental Examples section will change small components of the system.

The Base Example tutorial is organized as follows:

- *Example files*
- *Default Setup*
 - *Messages + Communication: pub sub*
 - *Simulator Integration: External JSON*
 - *Co-simulation Execution: helics_cli*
- *Questions and Help*

Example files

All files necessary to run the Base Example can be found in the [Fundamental examples repository](#):



The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- “runner” JSON to enable `helics_cli` execution of the co-simulation

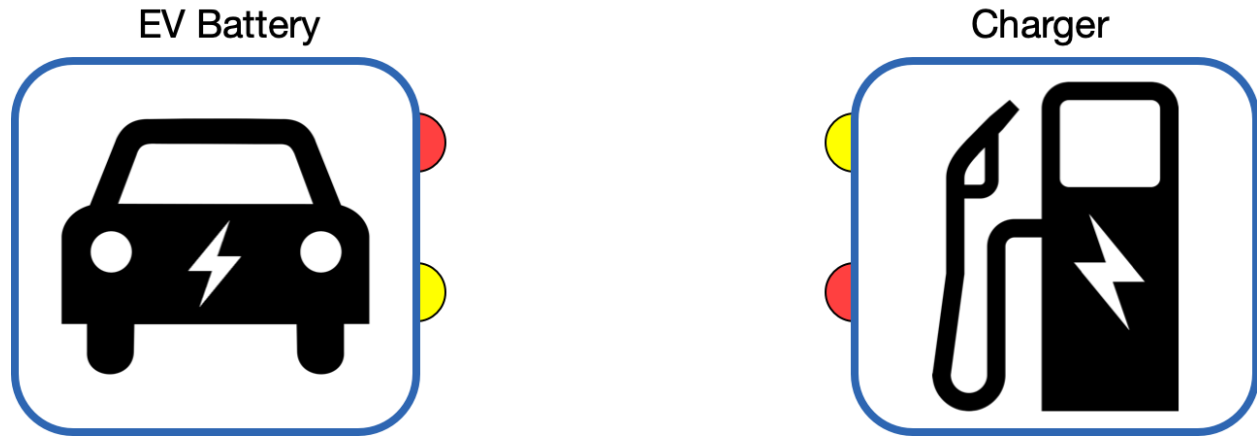
Default Setup

The default setup, used in the Base Example, integrates the federate configurations with external JSON files. The message and communication configurations are publications and subscriptions. We recommend launching the co-simulation with `helics_cli`. This section introduces federate configuration of publications (pubs) and subscriptions (subs) with JSON files and how to launch the co-simulation with `helics_cli`.

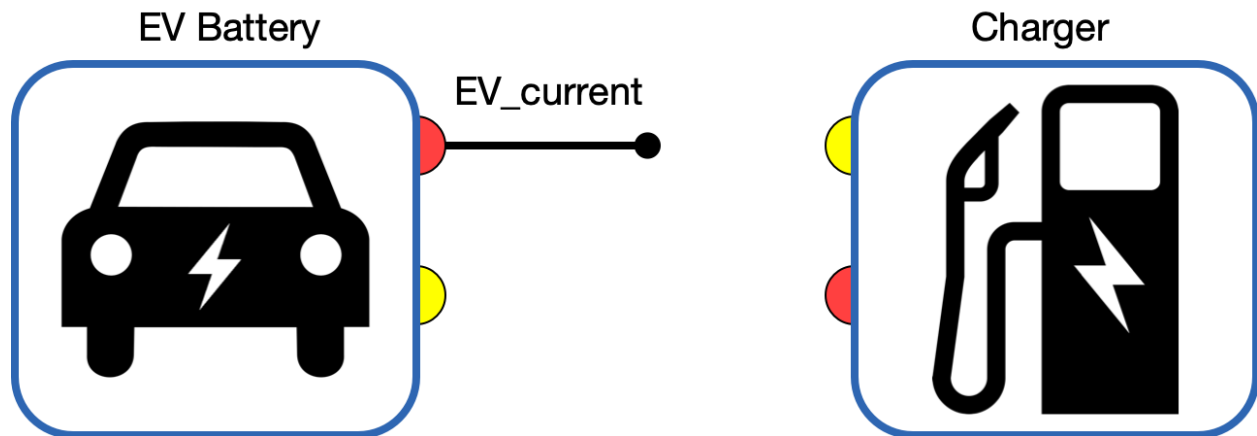
Messages + Communication: pub/sub

In the Base Example, the information being passed between the `Battery.py` federate and the `Charger.py` federate is the **voltage** applied to the battery, and the **current** measured across the battery and fed back to the charger. Voltage and current are both physical quantities, meaning that unless we act on these quantities to change them, they will retain their values. For this reason, in HELICS, physical quantities are called **values**. Values are sent via publication and subscription – a federate can publish its value(s), and another federate can subscribe this value(s).

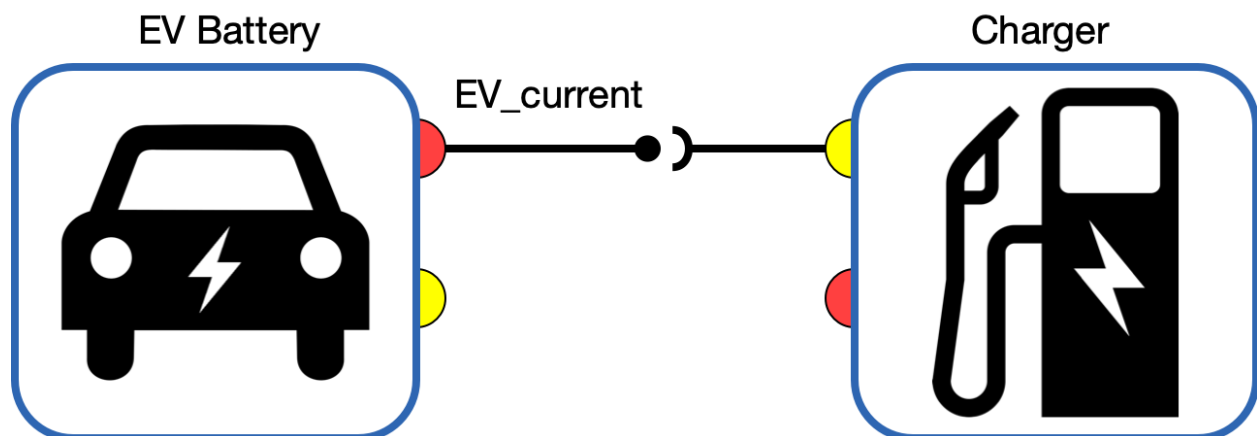
When configuring the communication passage between federates, it is important to connect the federate to the correct **handle**. In the image below, we have a Battery federate and a Charger federate. Each federate has a **publication** handle (red) and a **subscription** handle (yellow). The publication handle is also called the **output**, and the subscription handle the **input**. How are values passed between federates with pubs and subs?



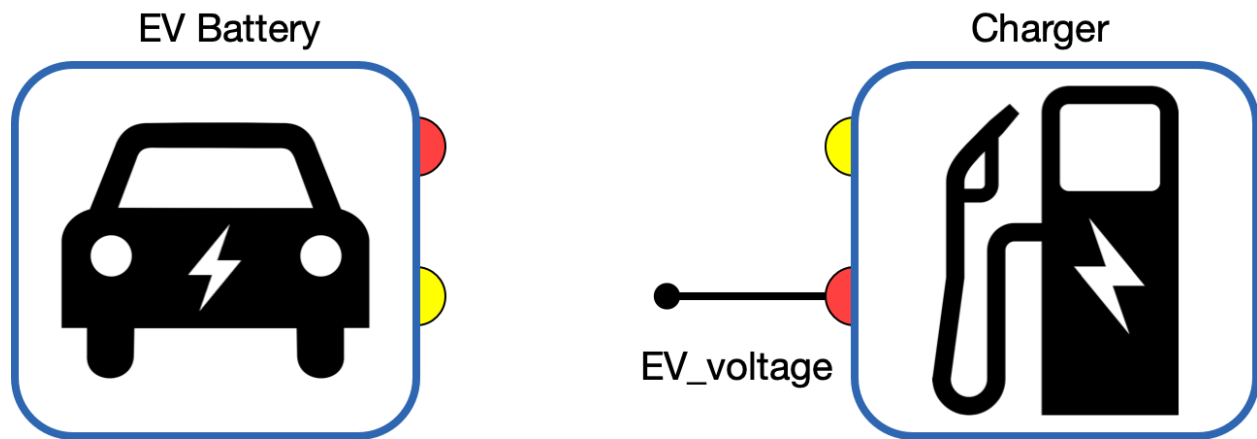
We have **named** the publication handle for the Battery federate `EV_current` to indicate the information being broadcast – we can also call the pub handle the **named output**. This is what the publication is doing – we are telling the Battery federate that we want to publish the `EV_current`. The full handle for the current is `Battery/EV_current` (within the JSON, this is also called the **key**).



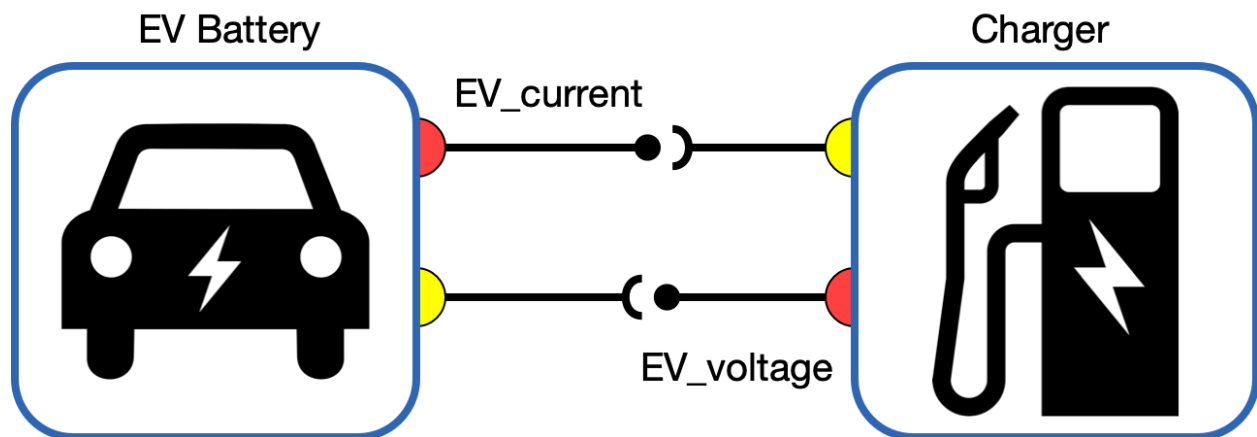
How does the current value get from the Battery federate's publication to the Charger federate? The Charger must subscribe to this publication handle – the Charger will subscribe to `Battery/EV_current`. The Charger subscription handle has not been given a name (e.g., `Charger/EV_current`), but it will receive **input** – the Charger subscription is a defined unnamed input with a targeted publication. In this example, we configure the target of the Charger subscription in the JSON to the publication handle name `Battery/EV_current`.



Thus far we have established that the Battery is publishing its current from the named handle Battery/EV_current and the Charger is subscribing to this named handle. The Charger is also sending information about values. The Charger federate will be publishing the voltage value from the Charger/EV_voltage handle (a named output).



In order to guarantee that the Battery federate receives the voltage value from the Charger federate, the Battery will have an unnamed input subscription which targets the Charger/EV_voltage handle.



With a better understanding of how we want to configure the pubs and subs, we can now move on to the mechanics of integrating the two simulators.

Simulator Integration: External JSON

Configuration of federates may be done with JSON files. Each federate will have its own configuration (“config”) file. It’s good practice to mirror the name of the federate with the config file. For example, the Battery.py federate will have a config file named BatteryConfig.json.

There are *extensive ways* to configure federates in HELICS. The BatteryConfig.json file contains the most common as defaults:

```
{
  "name": "Battery",
  "log_level": 1,
  "core_type": "zmq",
  "period": 60,
```

(continues on next page)

(continued from previous page)

```
"uninterruptible": false,
"terminate_on_error": true,
"wait_for_current_time_update": true,
"publications": [ ... ],
"subscriptions": [ ... ]
}
```

In this configuration, we have named the federate `Battery`, set the `log_level` to 1 (*what do loglevels mean and which one do I want?*), and set the `core_type` to `zmq` (*the most common*). The next four options control timing for this federate. The final options are for message passing.

This federate is configured with pubs and subs, so it will need an option to indicate the publication and the subscription configurations (for brevity, only the first pub and sub are printed below):

```
"publications": [
  {
    "key": "Battery/EV1_current",
    "type": "double",
    "unit": "A",
    "global": true
  },
  {...}
],
"subscriptions": [
  {
    "key": "Charger/EV1_voltage",
    "type": "double",
    "unit": "V",
    "global": true
  },
  {...}
]
```

This pub and sub configuration is telling us that the `Battery.py` federate is publishing in units of amps (A) for current from the named handle (key) `Battery/EV1_current`. This federate is also subscribing to information from the `Charger.py` federate. It has subscribed to a value in units of volts (V) at the named handle (key) `Charger/EV1_voltage`.

As discussed in *Register and Configure Federates*, the federate registration and configuration with JSON files in the python federate is done with one line of code:

```
fed = h.helicsCreateValueFederateFromConfig("BatteryConfig.json")
```

Recall that federate registration and configuration is typically done **before** entering execution mode.

Co-simulation Execution: `helics_cli`

At this point in setting up the Base Example co-simulation, we have:

1. Placed the necessary HELICS components in each federate program
2. Written the configuration JSON files for each federate

It's now time to launch the co-simulation with `helics_cli` ([install helics_cli here!](#)). This is accomplished by creating a **runner** JSON file. `helics_cli` allows the user to launch multiple simulations in one command line, which otherwise would have required multiple terminals.

The runner JSON for the Base Example is called `fundamental_default_runner.json`:

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 2 --loglevel=7",
      "host": "localhost",
      "name": "broker"
    },
    {
      "directory": ".",
      "exec": "python -u Charger.py 1",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py 1",
      "host": "localhost",
      "name": "Battery"
    }
  ],
  "name": "fundamental_default"
}
```

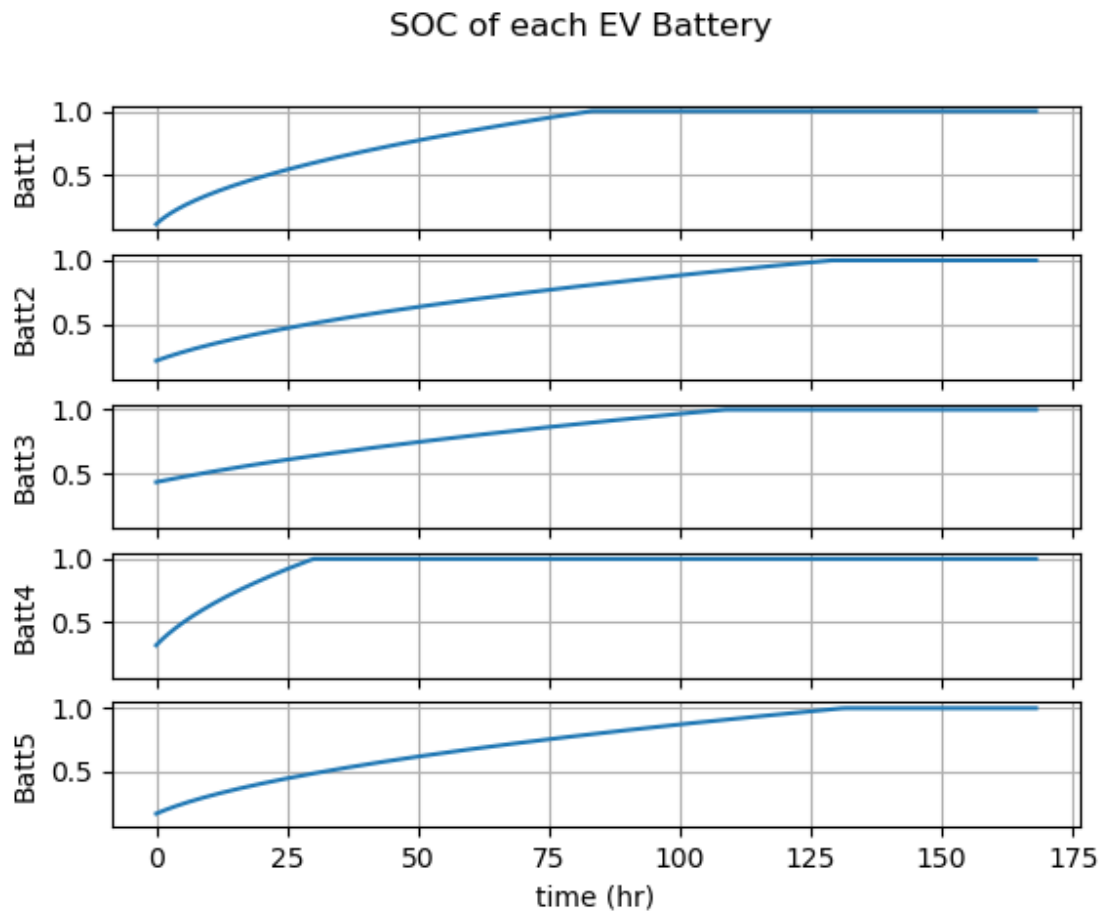
This runner tells `helics_broker` that there are three federates and to take a specific action for each federate:

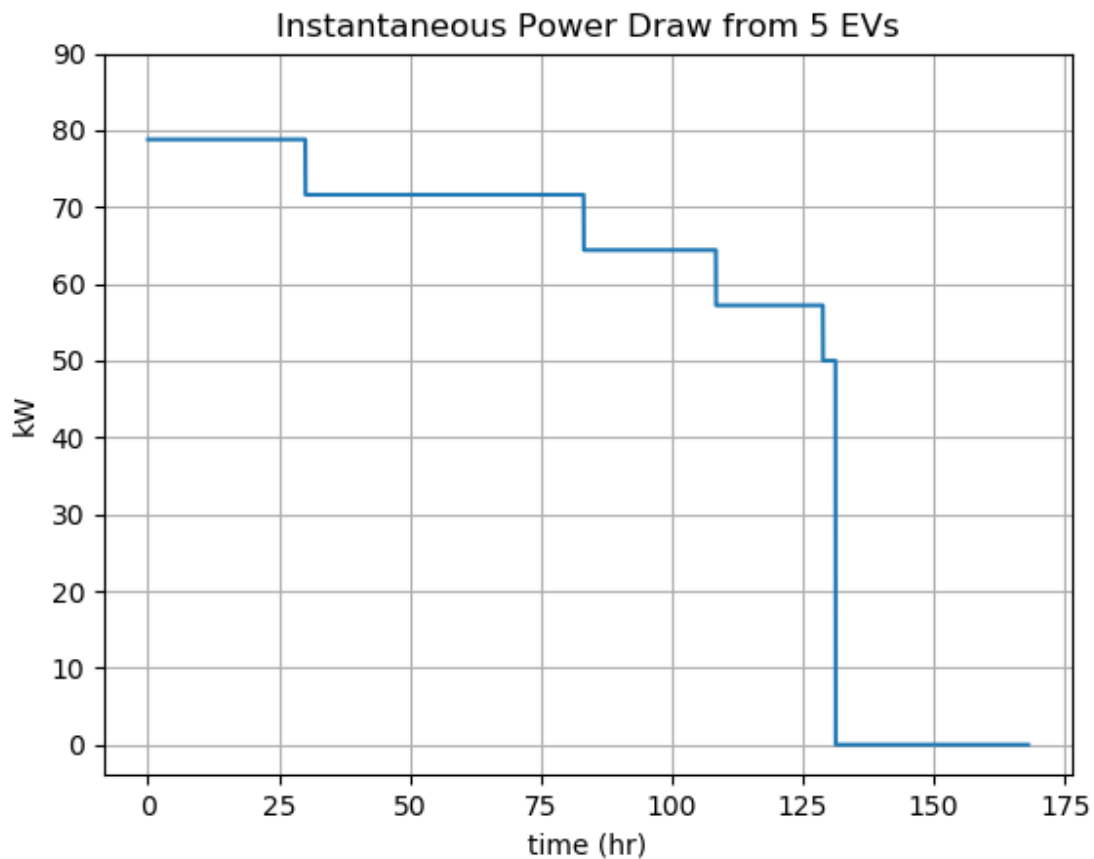
1. Launch `helics_broker` in the current directory: `helics_broker -f 2 --loglevel=7`
2. Launch the `Charger.py` federate in the current directory: `python -u Charger.py 1`
3. Launch the `Battery.py` federate in the current directory: `python -u Battery.py 1`

The final step is to launch our Base Example with `helics_cli` from the command line:

```
>helics run --path=fundamental_default_runner.json
```

If all goes well, this will reward us with two figures:





We can see the state of charge of each battery over the duration of the co-simulation in the first figure, and the aggregated instantaneous power draw in the second. As the engineer tasked with assessing the power needs for this charging garage, do you think you have enough information at this stage? If not, how would you change the co-simulation to better model the research needs?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Federate Integration with PyHELICS API

The Federate Integration Example extends the Base Example to demonstrate how to integrate federates using the HELICS API instead of JSON config files.

FEDERATE INTEGRATION

Simulator int → using HELICS API

Messages + Communication: pub/sub

Types of federates: python

Broker hierarchies: 1

This tutorial is organized as follows:

- *Computing Environment*
- *Example files*
- *Federate Integration using the PyHELICS API*
 - *Translation from JSON to PyHELICS API methods*
 - *Federate Integration with API calls*
 - *Dynamic Pub/Subs with API calls*
 - *Co-simulation Execution*
- *Questions and Help*

Computing Environment

This example was successfully run on Tue Nov 10 11:16:44 PST 2020 with the following computing environment.

- Operating System

```
$ sw_vers
ProductName:    Mac OS X
ProductVersion: 10.14.6
BuildVersion:   18G6032
```

- python version

```
$ python
Python 3.7.6 (default, Jan 8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

- python modules for this example

```
$ pip list | grep matplotlib
matplotlib 3.1.3
```

(continues on next page)

(continued from previous page)

```
$ pip list | grep numpy
numpy 1.18.5
```

If these modules are not installed, you can install them with

```
$ pip install matplotlib
$ pip install numpy
```

- `helics_broker` version

```
$ helics_broker --version
2.4.0 (2020-02-04)
```

- `helics_cli` version

```
$ helics --version
0.4.1-HEAD-ef36755
```

- `pyhelics` init file

```
$ python

>>> import helics as h
>>> h.__file__
'/Users/[username]/Software/pyhelics/helics/__init__.py'
```

Example files

All files necessary to run the Federate Integration Example can be found in the [Fundamental examples repository](#):

The screenshot shows the GitHub interface for the repository `GMLC-TDC / HELICS-Examples`. The breadcrumb navigation indicates the current path is `HELICS-Examples / user_guide_examples / fundamental / fundamental_integration /`. Below this, a commit by `allisonmcampbell` is shown with the message "initial commit of integration example using API". A table lists the files included in this commit:

File	Commit Message	Time
Battery.py	initial commit of integration example using API	4 days ago
Charger.py	initial commit of integration example using API	4 days ago
EVtoyrunner.json	initial commit of integration example using API	4 days ago

The files include:

- Python program for Battery federate
- Python program for Charger federate

- “runner” JSON to enable `helics_cli` execution of the co-simulation

Federate Integration using the PyHELICS API

This example differs from the Base Example in that we integrate the federates (simulators) into the co-simulation using the API instead of an external JSON config file. Integration and configuration of federates can be done either way – the biggest hurdle for most beginning users of HELICS is learning how to look for the appropriate API key to mirror the JSON config style.

For example, let’s look at our JSON config file of the Battery federate from the Base Example:

```
{
  "name": "Battery",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": true,
  "publications": [ ... ],
  "subscriptions": [ ... ]
}
```

We can see from this config file that we need to find API method to assign the `name`, `loglevel`, `coreType`, `period`, `uninterruptible`, `terminate_on_error`, `wait_for_current_time_update`, and `pub/subs`. In this example, we will be using the [PyHELICS API methods](#). This section will discuss how to translate JSON config files to API methods, how to configure the federate with these API calls in the co-simulation, and how to dynamically register publications and subscriptions with other federates.

Translation from JSON to PyHELICS API methods

Configuration with the API is done within the federate, where an API call sets the properties of the federate. With our Battery federate, the following API calls will set all the properties from our JSON file (except `pub/sub`, which we’ll cover in a moment). These calls set:

1. `name`
2. `loglevel`
3. `coreType`
4. Additional core configurations
5. `period`
6. `uninterruptible`
7. `terminate_on_error`
8. `wait_for_current_time_update`

```
h.helicsCreateValueFederate("Battery", fedinfo)
h.helicsFederateInfoSetIntegerProperty(fedinfo, h.HELICS_PROPERTY_INT_LOG_LEVEL, 1)
h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
```

(continues on next page)

(continued from previous page)

```

h.helicsFederateInfoSetTimeProperty(fedinfo, h.HELICS_PROPERTY_TIME_PERIOD, 60)
h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_UNINTERRUPTIBLE, False)
h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
h.helicsFederateInfoSetFlagOption(
    fedinfo, h.HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE, True
)

```

If you find yourself wanting to set additional properties, there are a handful of places you can look:

- [C++ source code](#): Do a string search for the JSON property. This can provide clarity into which enum to use from the API.
- [PyHELICS API methods](#): API methods specific to PyHELICS, with suggestions for making the calls pythonic.
- [Configuration Options Reference](#): API calls for C++, C, Python, and Julia

Federate Integration with API calls

We now know which API calls are analogous to the JSON configurations – how should these methods be called in the co-simulation to properly integrate the federate?

It's common practice to rely on a helper function to integrate the federate using API calls. With our Battery/Controller co-simulation, this is done by defining a `create_value_federate` function (named for the fact that the messages passed between the two federates are physical values). In `Battery.py` this function is:

```

def create_value_federate(fedinitstring, name, period):
    fedinfo = h.helicsCreateFederateInfo()
    h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
    h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
    h.helicsFederateInfoSetIntegerProperty(fedinfo, h.HELICS_PROPERTY_INT_LOG_LEVEL, 1)
    h.helicsFederateInfoSetTimeProperty(fedinfo, h.HELICS_PROPERTY_TIME_PERIOD, period)
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_UNINTERRUPTIBLE, False)
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
    h.helicsFederateInfoSetFlagOption(
        fedinfo, h.HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE, True
    )
    fed = h.helicsCreateValueFederate(name, fedinfo)
    return fed

```

Notice that we have passed three items to this function: `fedinitstring`, `name`, and `period`. This allows us to flexibly reuse this function if we decide later to change the name or the period (the most common values to change).

We create the federate and integrate it into the co-simulation by calling this function at the beginning of the program main loop:

```

fedinitstring = "--federates=1"
name = "Battery"
period = 60
fed = create_value_federate(fedinitstring, name, period)

```

What step created the value federate?

```
fed = h.helicsCreateValueFederate(name, fedinfo)
```

Notice that we pass to this API the `fedinfo` set by all preceding API calls.

Dynamic Pub/Subs with API calls

In the Base Example, we configured the pubs and subs with an external JSON file, where *each* publication and subscription between federate handles needed to be explicitly defined for a predetermined number of connections:

```
"publications":[
  {
    "key":"Battery/EV1_current",
    "type":"double",
    "unit":"A",
    "global": true
  },
  {...}
],
"subscriptions":[
  {
    "key":"Charger/EV1_voltage",
    "type":"double",
    "unit":"V",
    "global": true
  },
  {...}
]
```

With the PyHELICS API methods, you have the flexibility to define the connection configurations *dynamically* within execution of the main program loop. For example, in the Base Example we defined **five** communication connections between the Battery and the Charger, meant to model the interactions of five EVs each with their own charging port. If we want to increase or decrease that number using JSON configuration, we need to update the JSON file (either manually or with a script).

Using the PyHELICS API methods, we can register any number of publications and subscriptions. This example sets up pub/sub registration using for loops:

```
num_EVs = 5
pub_count = num_EVs
pubid = {}
for i in range(0, pub_count):
    pub_name = f"Battery/EV{i+1}_current"
    pubid[i] = h.helicsFederateRegisterGlobalTypePublication(
        fed, pub_name, "double", "A"
    )

sub_count = num_EVs
subid = {}
for i in range(0, sub_count):
    sub_name = f"Charger/EV{i+1}_voltage"
    subid[i] = h.helicsFederateRegisterSubscription(fed, sub_name, "V")
```

Here we only need to designate the number of connections to register in one place: `num_EVs = 5`. Then we register the publications using the `h.helicsFederateRegisterGlobalTypePublication()` method, and the subscriptions with the `h.helicsFederateRegisterSubscription()` method. Note that subscriptions are analogous to *inputs*, and as such retain similar properties.

Co-simulation Execution

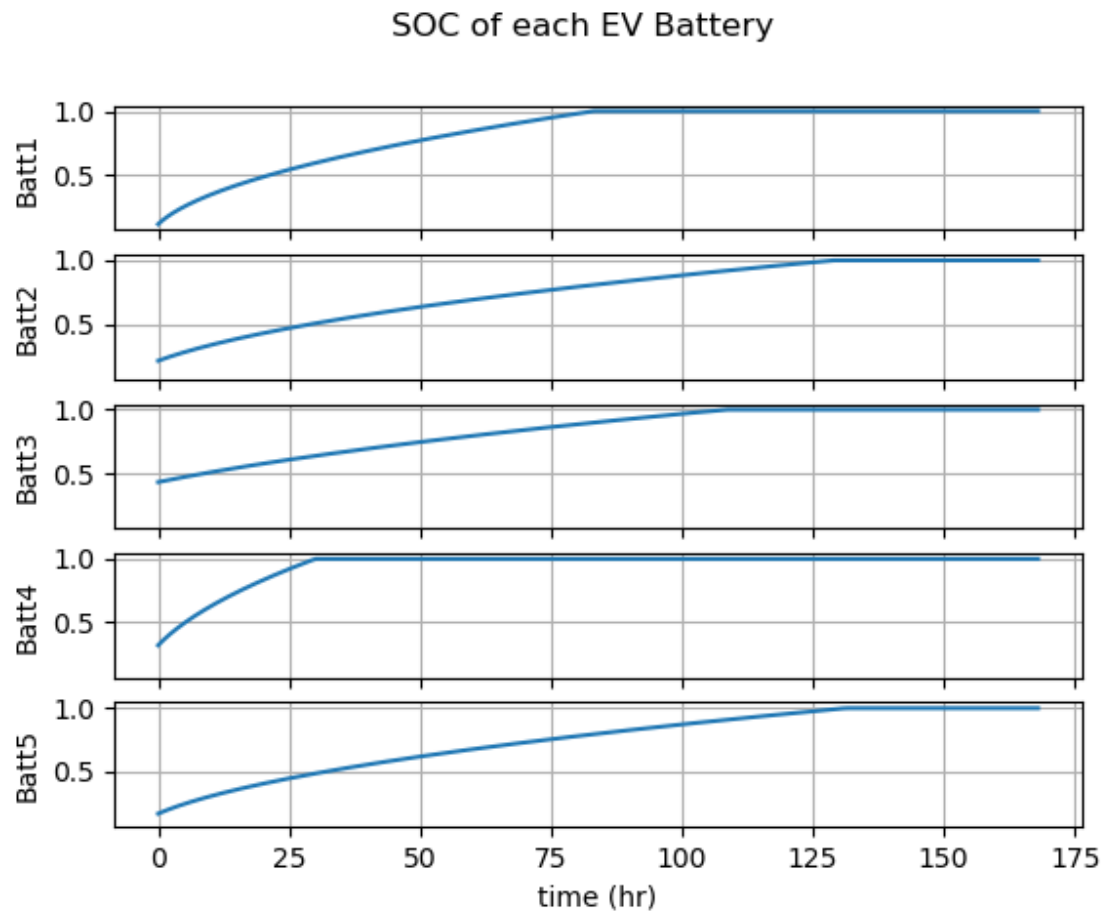
In this tutorial, we have covered how to integrate federates into a co-simulation using the PyHELICS API. Integration covers configuration of federates and registration of communication connections. Execution of the co-simulation is done the same as with the Base Example, with a runner JSON we sent to `helics_cli`. The runner JSON has not changed from the Base Example:

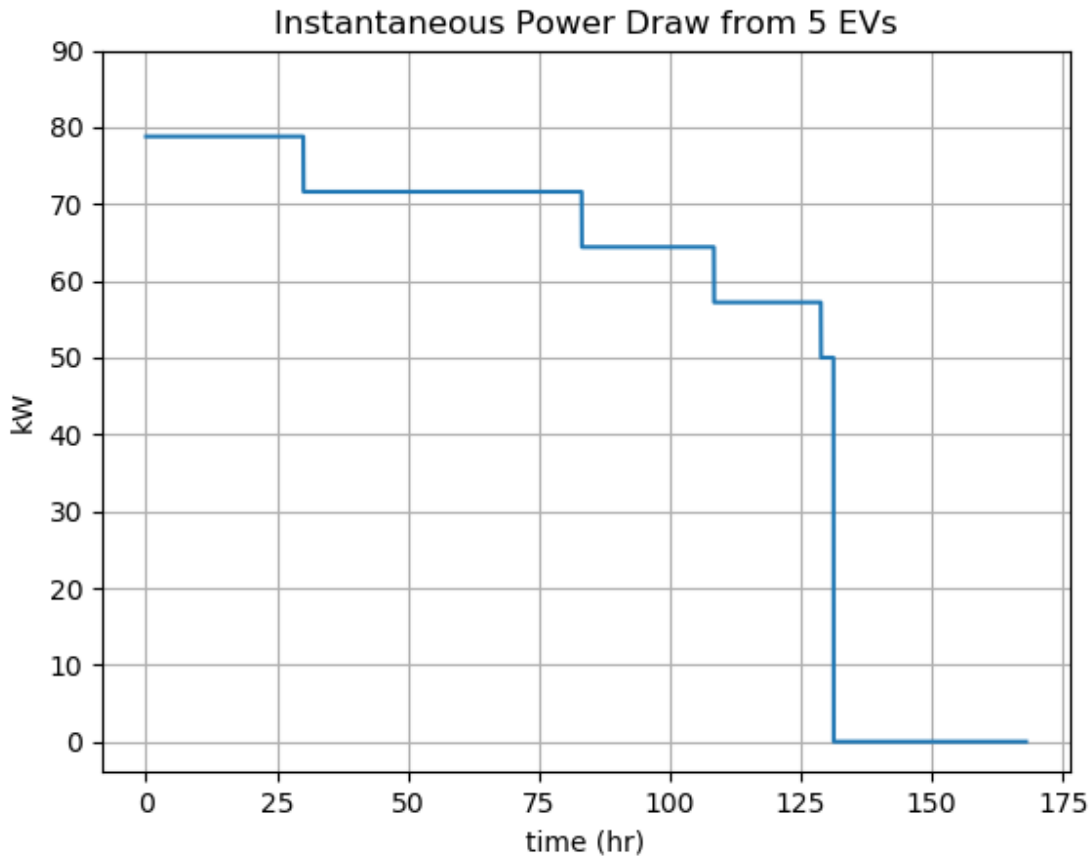
```
{
  "name": "fundamental_integration",
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=warning",
      "host": "localhost",
      "name": "broker"
    },
    {
      "directory": ".",
      "exec": "python -u Charger.py",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Controller.py",
      "host": "localhost",
      "name": "Controller"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py",
      "host": "localhost",
      "name": "Battery"
    }
  ]
}
```

Execute the co-simulation with the same command as the Base Example

```
>helics run --path=fundamental_integration_runner.json
```

This results in the same output; the only thing we have changed is the method of configuring the federates and integrating them.





If your output is not the same as with the Base Example, it can be helpful to pinpoint source of the difference – have you used the correct API method?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Federate Message + Communication Configuration

Endpoint Federates

The Federate Message + Communication Configuration Example extends the Base Example to demonstrate how to register federates which send messages from/to endpoints instead of values from/to pub/subs.

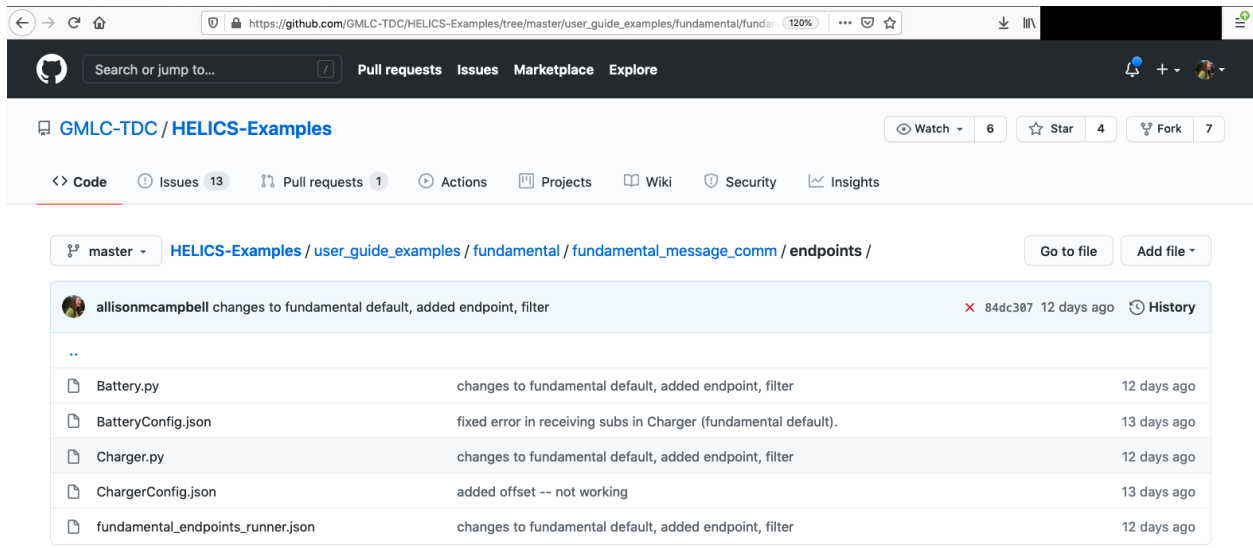
This tutorial is organized as follows:

- *Example files*
- *Federate Communication with Endpoints*

- *When to use pub/subs vs endpoints*
- *Translation from pub/sub to endpoints*
 - * *Config Files*
 - * *Simulators*
- *Co-simulation Execution*
- *Questions and Help*

Example files

All files necessary to run the Federate Integration Example can be found in the [Fundamental examples repository](#):



The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- “runner” JSON to enable `helics_cli` execution of the co-simulation

Federate Communication with Endpoints

There are two fundamental cases where you may find yourself using endpoints to send messages.

1. The federate is modeling communication of information (typically as a string)
2. The federate is incorrectly modeling communication of physical dynamics

What’s the difference? In the *Base Example*, the federation consists of two “value” federates – one passes its current, the other passes a voltage. The two depend on this information. The Battery says to the Charging port, “I have a starting current!”, to which the Charger responds, “Great, here’s your voltage!” These two **value** federates must be coupled with pub/subs, because they are linked by a physical system.

However, as the author of a HELICS co-simulation, there is nothing preventing you from connecting these two federates with endpoints in place of pub/subs. The co-simulation will give the same results in simple cases, but be wary of taking this type of short cut – resurrecting code which passes information in the incorrect way may introduce nefarious

results. The recommended approach is to register federates modeling **physics** as **value federates**, and those modeling *information* as *message federates*.

Casting this guidance to the wind, this example walks through how to set up the Base Example (which passes current and voltage) as **message federates** – landing this example squarely in situation #2 above. This is just for demonstration purposes, and this is the only example in the documentation which violates best practices.

When to use pub/subs vs endpoints

The easiest way to determine whether you should use pub/subs vs endpoints to connect your federates is to ask yourself: “Does this message have a physical unit of measurement?” As noted above, the Battery-Charger federation **does** model physical components, and the config files make this clear with the inclusion of units:

BatteryConfig.json:

```
"publications": [
  {
    "key": "Battery/EV1_current",
    "type": "double",
    "unit": "A",
    "global": true
  },

```

```
"subscriptions": [
  {
    "key": "Charger/EV1_voltage",
    "type": "double",
    "unit": "V",
    "global": true
  },

```

ChargerConfig.json:

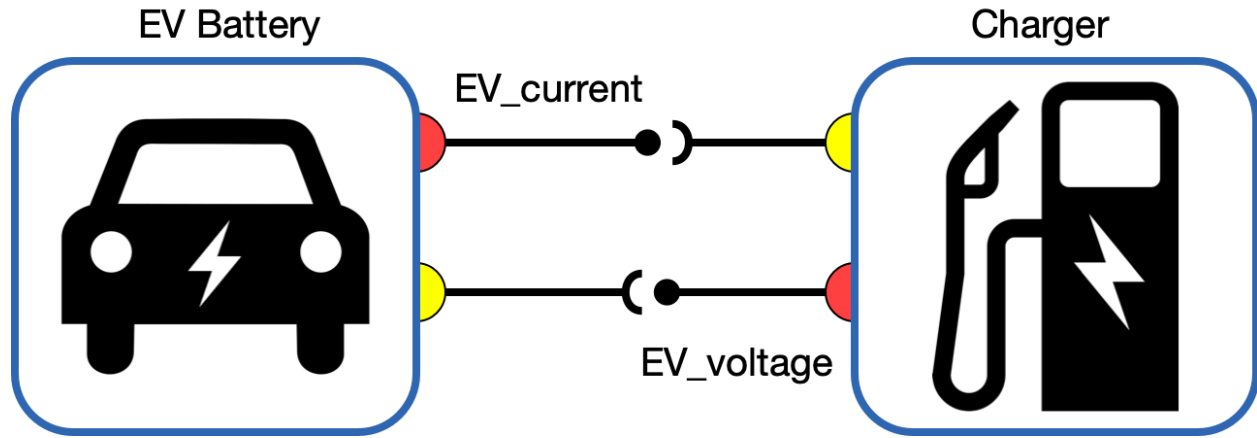
```
"publications": [
  {
    "key": "Charger/EV1_voltage",
    "type": "double",
    "unit": "V",
    "global": true
  },

```

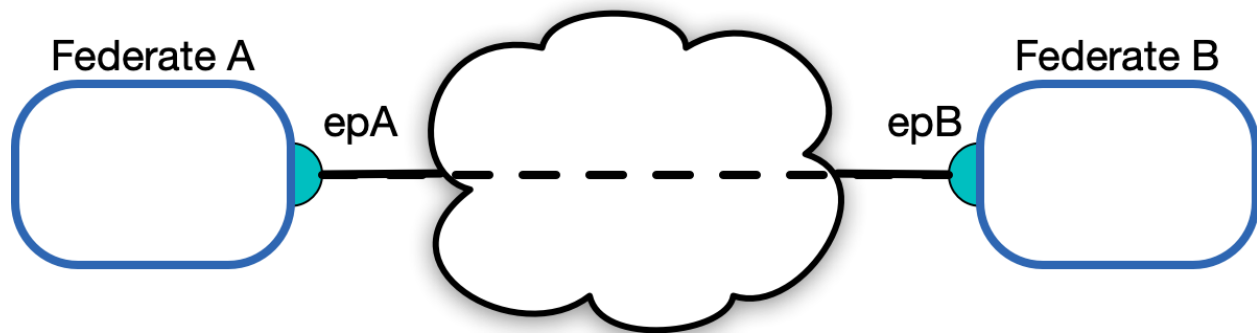
```
"subscriptions": [
  {
    "key": "Battery/EV1_current",
    "type": "double",
    "unit": "A",
    "global": true
  },

```

With this pub/sub configuration, we have established a **direct** communication link between the Battery and Charger:



If we accept that the information being passed between the two is not physics-based, then the communication link only depends on each federate having an endpoint:



In departure from the directly-coupled communication links of pub/subs, messages sent from **endpoints** can be intercepted, delayed, or picked up by any federate. In that sense, communication via pub/subs can be thought of as sealed letters sent via pneumatic tubes, and messages sent via endpoints as a return-address labeled letter sent into the postal service system.

You might ask yourself: “How does HELICS know where to send the message?” There are ways to set this up as default. Before we dive into the code, it’s important to understand the following about messages and endpoints:

1. Endpoints send messages as encoded strings
2. Endpoints can have default destinations, but this is not required
3. Endpoints should not be used to model physics
4. Messages sent from endpoints are allowed to be corrupted (see [Filters!](#))
5. Messages sent from endpoints do not show up on a HELICS `dependency_graph` (A `dependency_graph` is a graph of dependencies in a federation. Because pub/subs have explicit connections, HELICS can establish when the information arrives through a `dependency_graph`. See [Queries](#) for more information.)

Translation from pub/sub to endpoints

We are throwing caution to the wind using endpoints to model the physics in the *Base Example*. Changes need to be made in the config files and the simulator code.

Config Files

As with the Base Example, configuration can be done with JSON files. The first change we need to make is to replace the publications and subscriptions with endpoints:

BatteryConfig.json:

```
"endpoints":[
  {
    "key":"Battery/EV1_current",
    "destination":"Charger/EV1_voltage",
    "global": true
  },
  {...}
]
```

ChargerConfig.json:

```
"endpoints":[
  {
    "key":"Charger/EV1_voltage",
    "destination":"Battery/EV1_current",
    "global": true
  },
  {...}
]
```

If you have run the Base Example, you will have seen that the information passed between the federates occurs at the same HELICS time; both federates have "period": 60 in their config files. Recall from the *Configuration Options Reference* that the period controls the minimum time resolution for a federate.

We have a federation sending messages at the same time ("period": 60), and HELICS doesn't know which message arrives first. We need to introduce an offset to the config file of one of the federates to force it to wait until the message has been received. We also need to keep "uninterruptible": false, so that the federate will be granted the time at which it has received a message (which will be "period": 60).

The order of operation is:

Step	HELICS Time	Charger	Battery
1	0	requests time = 60, granted time = 60	requests time = 70
2	60	sends message to default destination "Battery/EV1_current"	granted time = 60 because message has arrived
3	60		sends message to default destination "Charger/EV1_voltage"

Introducing the offset into the config file (along with "uninterruptible": false) instructs HELICS about the dependencies. These adjustments are:

BatteryConfig.json:

```
{
  "name": "Battery",
  "loglevel": 7,
  "coreType": "zmq",
  "period": 60.0,
  "offset": 10.0,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": [...]
```

ChargerConfig.json:

```
{
  "name": "Charger",
  "loglevel": 7,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": [...]
```

Notice that we have only introduced an `offset` into the Battery config file, as we have set up the federates such that the Battery is waiting for information from the Charger.

Simulators

The simulators in this co-simulation (*.py) must be edited from the Base Example to register endpoints (instead of pub/subs) and ensure that messages are sent and received.

Battery

First let's make changes to `Battery.py`. In the Registration Step, we need to register the endpoints and get the endpoint count:

```
fed = h.helicsCreateMessageFederateFromConfig("BatteryConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
print(f"Created federate {federate_name}")

end_count = h.helicsFederateGetEndpointCount(fed)
logging.debug(f"\tNumber of endpoints: {end_count}")

# Diagnostics to confirm JSON config correctly added the required
# endpoints
endidx = {}
for i in range(0, end_count):
    endidx[i] = h.helicsFederateGetEndpointByIndex(fed, i)
    end_name = h.helicsEndpointGetName(endidx[i])
    logger.debug(f"\tRegistered Endpoint ---> {end_name}")
```

After entering Execution Mode but before the Time Loop begins, we need to extract the offset value:

```
update_offset = int(h.helicsFederateGetTimeProperty(fed, h.helics_property_time_offset))
```

And add that offset to requested_time:

```
requested_time = grantedtime + update_interval + update_offset
```

The next largest difference with implementing communication between simulators with endpoints vs pub/subs comes from the lack of innate message dependency, as described above with the `dependency_graph`. (Which can be accessed for pub/subs with a *query*.) Best practice for handling message receipt is to check if a message is waiting to be retrieved for an endpoint. The following code replaces `charging_voltage = h.helicsInputGetDouble((subid[j]))` from the Base Example (we are looping over `end_count`, the number of endpoints for this federate):

```
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    charging_voltage = float(h.helicsMessageGetString(msg))
```

If we want to know who sent the message (which can be helpful for both debugging and simplifying code), we invoke:

```
source = h.helicsMessageGetOriginalSource(msg)
```

An alternative to using `h.helicsMessageGetOriginalSource(msg)` is to set a default destination in the JSON config file. Use of both can help with debugging.

The `Battery.py` simulator takes the `charging_voltage` from the `Charger.py` simulator and calculates the `charging_current` to send back. Sending messages to a default destination is then done with:

```
h.helicsEndpointSendBytesTo(endid[j], "", str(charging_current))
```

Where the `""` can also be replaced with a string for the desired destination – we can check `""` against `source` to confirm the messages are going to their intended destinations.

Charger

As with the `Battery.py` simulator, we need to Register the `Charger` federate as a Message Federate and get the endpoint ids:

```
fed = h.helicsCreateMessageFederateFromConfig("ChargerConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
print(f"Created federate {federate_name}")
end_count = h.helicsFederateGetEndpointCount(fed)
logging.debug(f"\tNumber of endpoints: {end_count}")

# Diagnostics to confirm JSON config correctly added the required
# endpoints
endid = {}
for i in range(0, end_count):
    endid[i] = h.helicsFederateGetEndpointByIndex(fed, i)
    end_name = h.helicsEndpointGetName(endid[i])
    logger.debug(f"\tRegistered Endpoint ---> {end_name}")
```

The next difference with the Base Example `Charger.py` simulator is in sending the initial voltage to `Battery Federate`:

```
for j in range(0, end_count):
    message = str(charging_voltage[j])
    h.helicsEndpointSendBytesTo(endid[j], "", message.encode()) #
```

Notice that we are sending the message to the default destination with "". We cannot use `h.helicsMessageGetOriginalSource(msg)`, as no messages have been received by the Charger Federate yet.

Within the Time Loop, we change the message receipt component in the same way as the `Battery.py` simulator, where `h.helicsInputGetDouble((subid[j]))` is replaced with:

```
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    charging_current[j] = float(h.helicsMessageGetString(msg))
```

There's one final difference. Which API do we call to send the message to the Battery Federate?

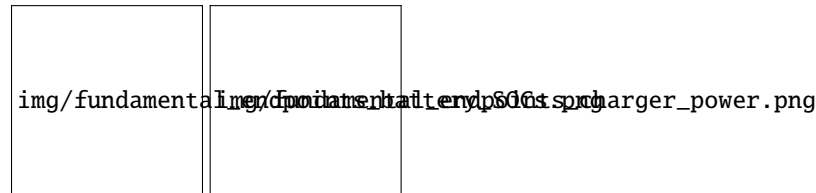
```
# Send message of voltage to Battery federate
h.helicsEndpointSendBytesTo(endid[j], "", f'{charging_voltage[j]:4f}').
.encode() #
```

Co-simulation execution

We run the co-simulation just as before in the Base Example – the `runner.json` is exactly the same:

```
>helics run --path=fundamental_endpoints_runner.json
```

And we get these figures:



Armed now with the knowledge of endpoints and messages, how could you change the research question?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Combination Federation

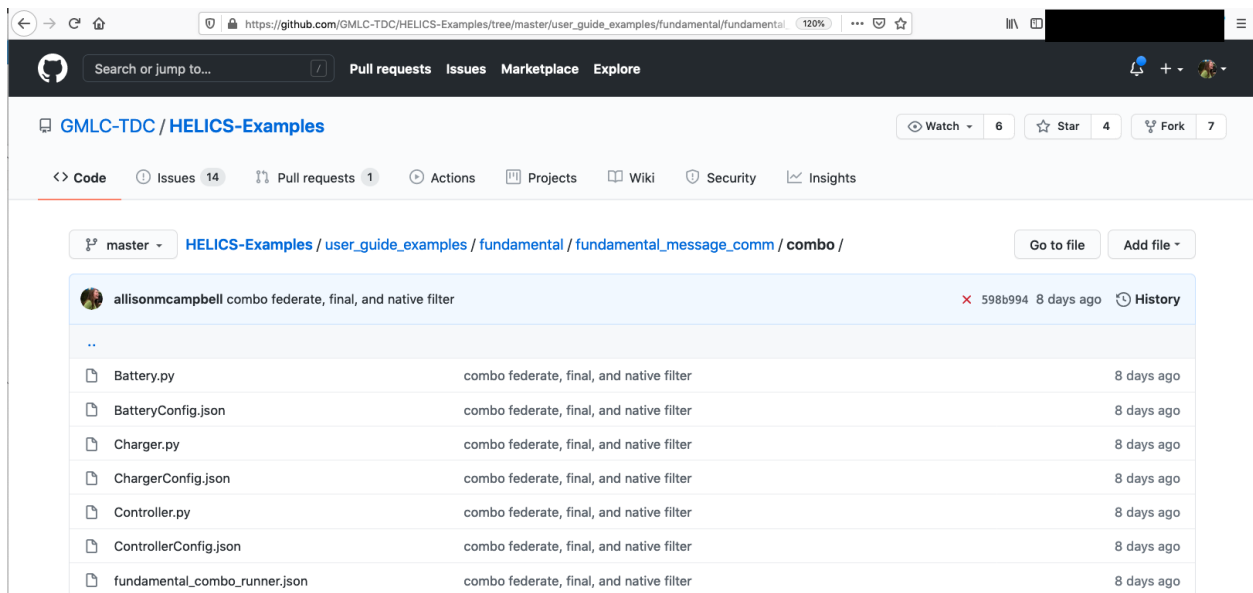
The Federate Message + Communication Configuration Example extends the Base Example to demonstrate how to register federates which can send/receive messages from endpoints and values from pub/subs. This example assumes the user has already worked through the [Endpoints Example](#).

This tutorial is organized as follows:

- [Example files](#)
- [Combination Federates](#)
 - [When to use pub/subs vs endpoints](#)
 - [Translation from pub/sub to endpoints](#)
 - [Co-simulation Execution](#)
- [Questions and Help](#)

Example files

All files necessary to run the Federate Integration Example can be found in the [Fundamental examples repository](#):



- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- “runner” JSON to enable `helics_cli` execution of the co-simulation

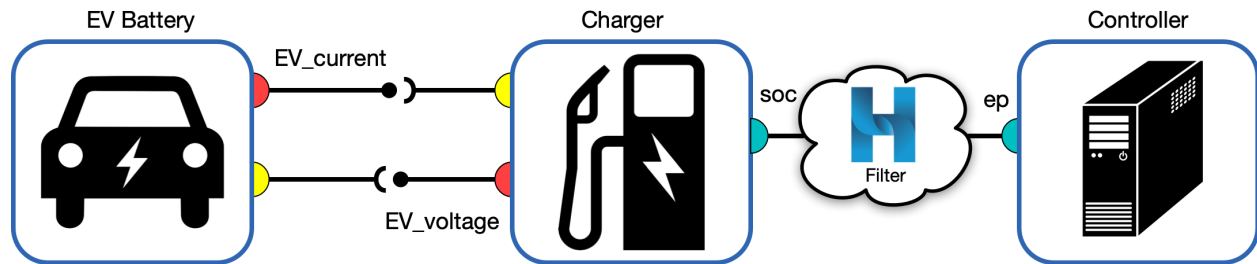
Combination Federates

A quick glance at the [Fundamental examples repository](#) on github will show that almost all these introductory examples are mocked up with two federates. These two federates pass information back and forth, and the examples show different ways this can be done.

This is the only example in the Fundamental series which models three federates – it is also exactly the same model as the [Base Example](#) in the Advanced series. Why are we introducing a third federate?

In the [Endpoints Example](#), we learned how to pass messages between two federates. The problem with this setup – which we will resolve in this example – is that **physical values** should not be modeled with messages/endpoints (see [the example](#) for a reminder). We introduce a third federate – a **combination federate** – to preserve the handling of physical values among *value federates* and allow for nuanced message passing (and interruption) among *message federates*. The key with combo federates is that they are the go-between for these two types. Combination federates can update (send) values *and* intercept messages. (For a refresher on values and messages, see the section on [Types of Federates](#). In brief: values have a physics-based unit, and messages are typically strings).

Here is our new federation of three federates:



We have:

- Battery (**value federate**: passes values with Charger through pub/subs)
- Charger (**combo federate**: passes values with Battery, passes messages with Controller)
- Controller (**message federate**: passes messages with Charger through endpoints)

Redistribution of Federate Roles

The full co-simulation is still asking the same question: “What is the expected instantaneous power draw from a dedicated EV charging garage?” With the introduction of a *Controller* federate, we now have additional flexibility in addressing the nuances to this question. For example, the charging controller does not have direct knowledge of the instantaneous current in the battery – the onboard charger needs to estimate this in order to calculate the EV’s state of charge. Let’s walk through the roles of each federate.

Battery

The Battery federate operates in the same way as in the Base Example. The only difference is that it is now allowed to request a new battery when an existing one is deemed to have a full SOC. This information is in the `charging_voltage` value from the Battery’s subscription to the Charger; if the Charger applies zero voltage, this means the Battery can no longer charge. The Battery federate selects a new battery randomly from three sizes – small, medium, and large – and assigns a random SOC between 0% and 80%.

There are no differences in the config file. As in the Base Example, the Battery federate logs and plots the internally calculated SOC over time at each charging port.

Charger

The Charger federate is now a combination federate – it will communicate via pub/subs with the Battery, and via endpoints with the Controller. This difference from the Base Example Charger is seen in the config file; in addition to the pub/subs with the Battery, there are now also endpoints. Notice that the default destination for each of these named endpoints is the same – there is one controller for all the charging ports.

```
"endpoints": [
  {
    "name": "Charger/EV1.soc",
    "destination": "Controller/ep",
    "global": true
  },
]
```

Since this federate also communicated via endpoints, we need to register them along with the existing pub/subs:

```
##### Registering federate from json #####
fed = h.helicsCreateCombinationFederateFromConfig("ChargerConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
end_count = h.helicsFederateGetEndpointCount(fed)
logger.info(f"\tNumber of endpoints: {end_count}")
sub_count = h.helicsFederateGetInputCount(fed)
logger.info(f"\tNumber of subscriptions: {sub_count}")
pub_count = h.helicsFederateGetPublicationCount(fed)
logger.info(f"\tNumber of publications: {pub_count}")
```

The Charger federate is gaining the new role of *estimating the Battery's current* and shifting the role of *deciding when to stop charging* to the Controller federate.

The Charger federate estimates the Battery federate's current with a new helper function call `estimate_SOC`. The Charger does not know the exact SOC of the Battery; it must estimate the SOC from the effective resistance, which is a function of applied voltage (from the Charger) and the measured current (from the Battery). This is the same function as used in the Battery federate, but with noise added to the measurement of the current.

```
def estimate_SOC(charging_V, charging_A):
    socs = np.array([0, 1])
    effective_R = np.array([8, 150])
    mu = 0
    sigma = 0.2
    noise = np.random.normal(mu, sigma)
    measured_A = charging_A + noise
    measured_R = charging_V / measured_A
    SOC_estimate = np.interp(measured_R, effective_R, socs)

    return SOC_estimate
```

This function is called after the Charger has received the charging current from the Battery federate and needs to update the SOC; if the current is not zero, the Charger estimates the SOC with the inclusion of measurement error on the current. This allows the co-simulation to model the separation of knowledge of the physics between the two federates: the Battery knows its internal current, but the on board Charger must estimate it.

If the current received from the Battery federate is zero, this means that we have plugged a new EV into the charging port and we need to determine the voltage to apply with the Charger. This is accomplished by calling `get_new_EV(1)` and `calc_charging_voltage()`. `get_new_EV(1)` is a helper function which selects the charging level (1, 2, or 3)

based on a set probability distribution and `calc_charging_voltage()` gives the applied voltage for that level. Once a “new EV” (the charging level) has been retrieved, the federate is assigned a SOC of 0 as an initial estimate prior to measuring the current.

The estimated SOC is sent to the Controller every 15 minutes – this mimics an on board charging agent regularly pinging the charging port to confirm if it should continue charging:

```
# Send message to Controller with SOC every 15 minutes
if grantedtime % 900 == 0:
    h.helicsEndpointSendBytesTo(endid[j], "", f"{currentsoc[j]:4f}".encode())
```

The Charger federate is allowed to be interrupted if there is a message from the Controller.

```
# Check for messages from EV Controller
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    instructions = h.helicsMessageGetString(msg)
```

The Charger will receive a message every 15 minutes as well, however it will only change actions if it is told to stop charging. When this happens, the Charger “disengages” from the charging port by applying zero voltage to the Battery.

```
if int(instructions) == 0:
    # Stop charging this EV
    charging_voltage[j] = 0
    logger.info(f"\tEV full; removing charging voltage")
```

Controller

The Controller is a new federate whose role is to decide whether to keep charging an EV based. This decision is based entirely on the estimated SOC calculated by the Charger. Since this decision logic is simple and can be applied to all the EVs modeled by the federation, we can set up the config file with one endpoint:

```
"endpoints": [
    {
        "name": "Controller/ep",
        "global": true
    }
]
```

Note that there is no default destination – the Controller will *respond* to a request for instructions from the Charger. This is accomplished by calling the `h.helicsMessageGetOriginalSource()` API:

```
while h.helicsEndpointHasMessage(endid):

    # Get the SOC from the EV/charging terminal in question
    msg = h.helicsEndpointGetMessage(endid)
    currentsoc = h.helicsMessageGetString(msg)
    source = h.helicsMessageGetOriginalSource(msg)
```

And then sending the message to this source:

```
message = str(instructions)
h.helicsEndpointSendBytesTo(endid, source, message.encode())
```


The Controller federate only operates when it receives a message – it is a *passive* federate. This can be set up by:

1. Initializing the start time of the federate to `h.HELICS_TIME_MAXTIME`:

```
fake_max_time = int(h.HELICS_TIME_MAXTIME)
starttime = fake_max_time
logger.debug(f"Requesting initial time {starttime}")
grantedtime = h.helicsFederateRequestTime(fed, starttime)
```

2. Allow the federate to be interrupted and set a minimum `timedelta` (`ControllerConfig.json`):

```
{
  "name": "Controller",
  ...
  "timedelta": 1,
  "uninterruptible": false,
  ...
}
```

3. Only execute an action when there is a message:

```
while h.helicsEndpointHasMessage(endid):
    pass # placeholder for loop body
```

4. Re-request the `h.HELICS_TIME_MAXTIME` after a message has been received:

```
grantedtime = h.helicsFederateRequestTime(fed, fake_max_time)
```

The message the Controller receives is the SOC estimated by the Charger. If the estimated SOC is greater than 95%, the Controller sends the message back to stop charging.

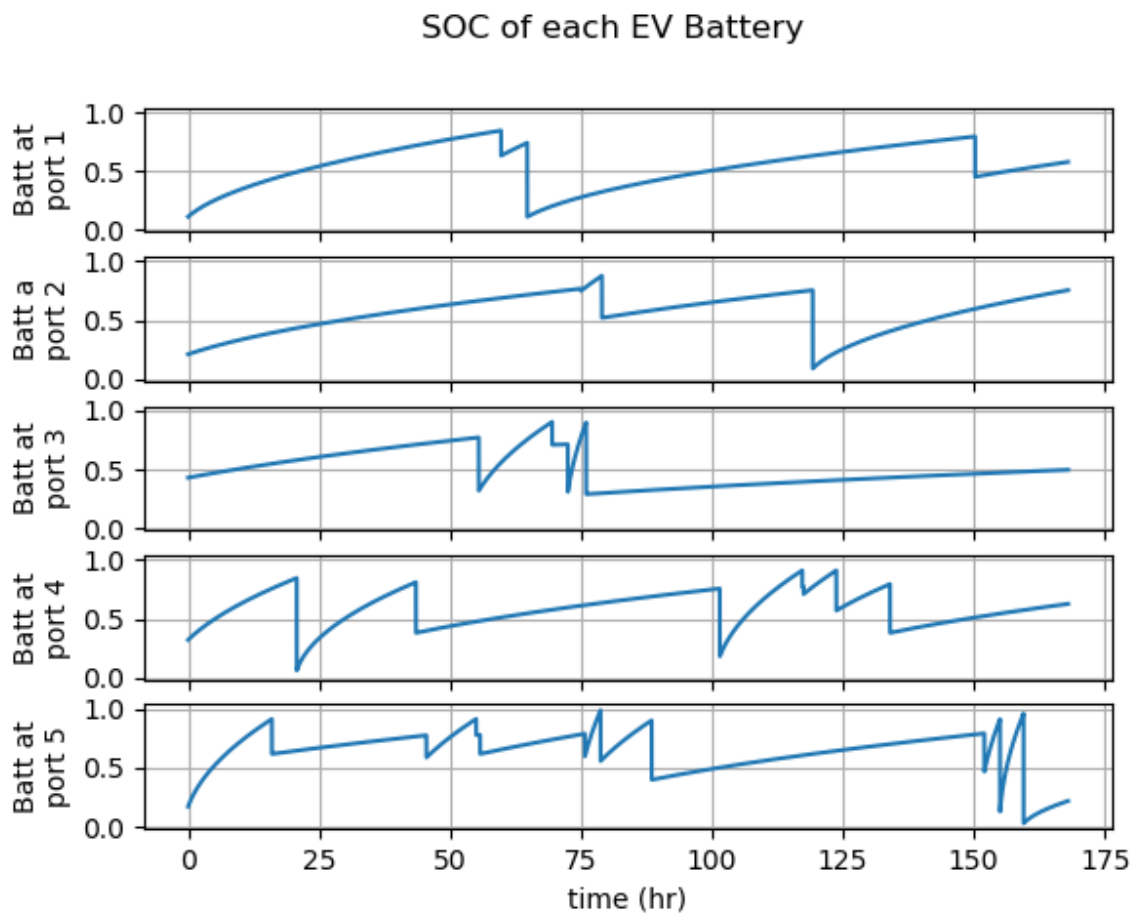
```
soc_full = 0.95
if float(currentsoc) <= soc_full:
    instructions = 1
else:
    instructions = 0
```

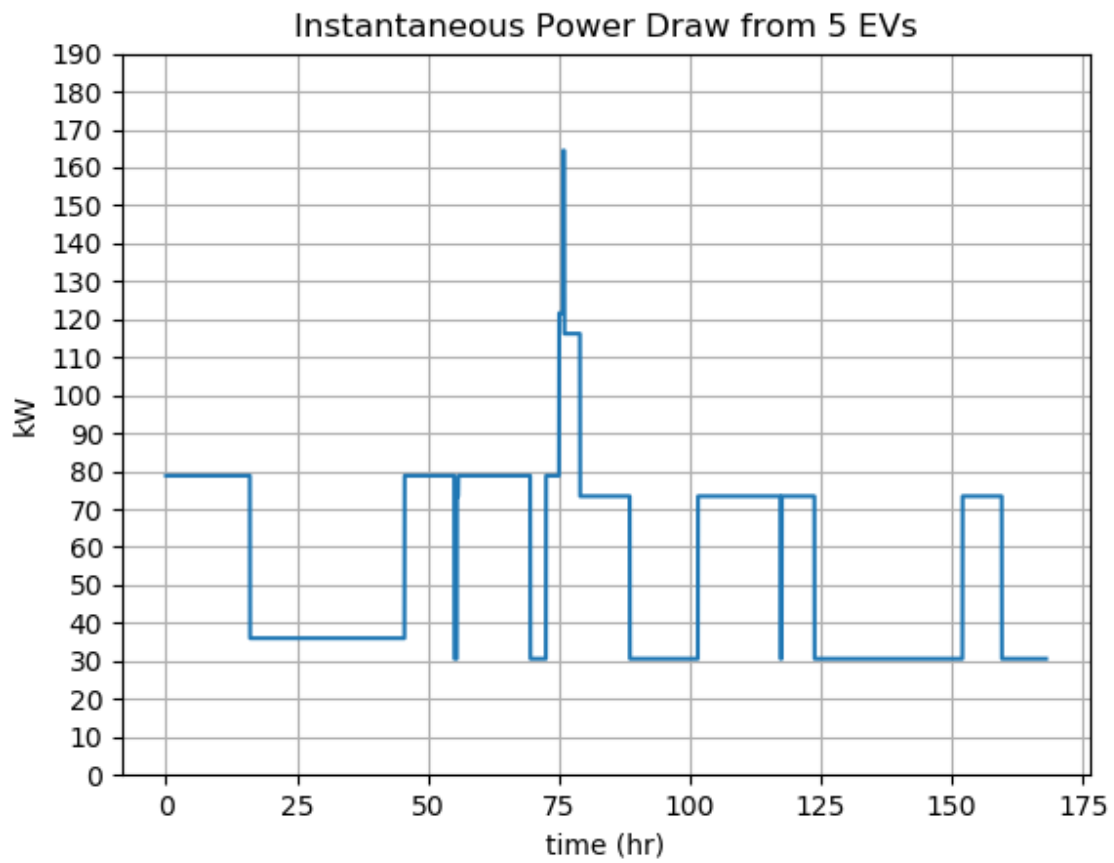
Co-simulation execution

With these three federates – Battery, Charger, and Controller – we have partitioned the roles into the most logical places. Execution of this co-simulation is done as before with `helics_cli`:

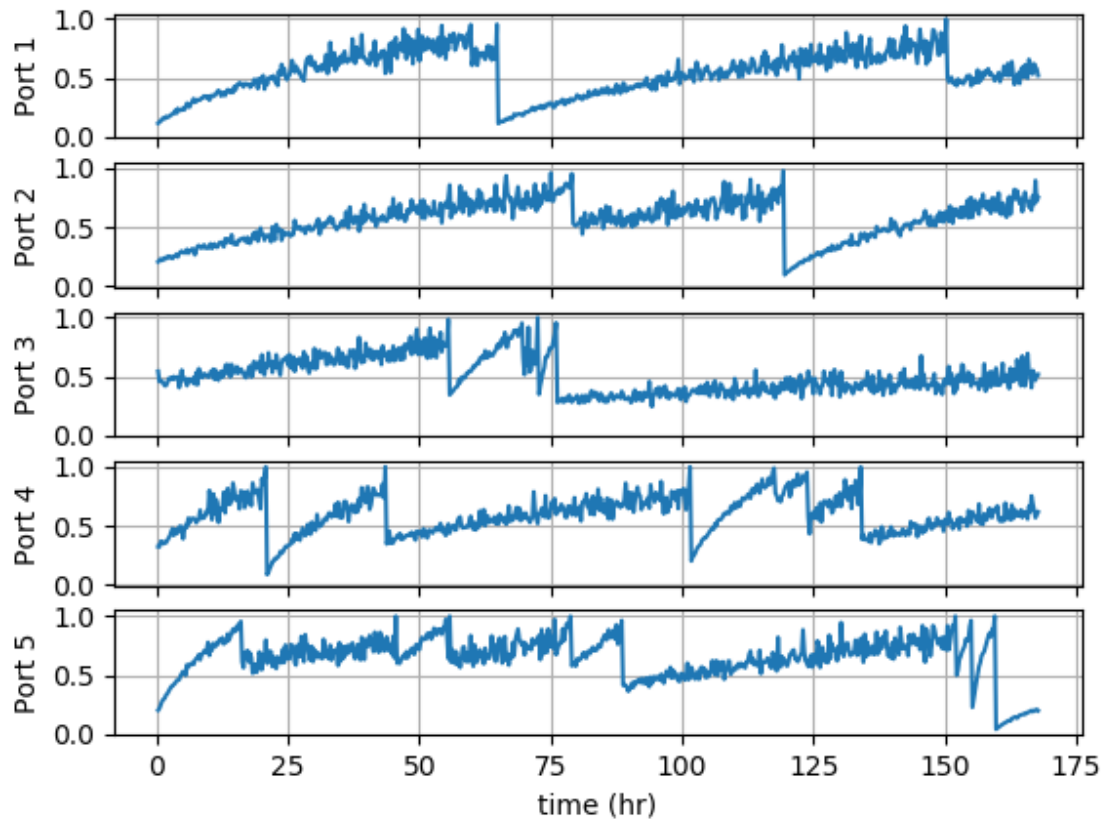
```
helics run --path=fundamental_combo_runner.json
```

The resulting figures show the actual on board SOC at each EV charging port, the instantaneous power draw, and the SOC estimated by the on board charger.





SOC at each charging port



Note that we have made a number of simplifying assumptions in this analysis:

- There will always be an EV waiting to be charged (the charging ports are never idle).
- There is a constant number of charging ports – we know what the power draw will look like given a static number of ports, but we do not know the underlying demand for power from EVs.
- The equipment which ferries the messages between the Charger and the Controller never fails – we haven't incorporated *Filters*.

How would you model an unknown demand for vehicle charging? How would you model idle charging ports? What other simplifications do you see that can be addressed?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Combination Federation with Native HELICS Filters

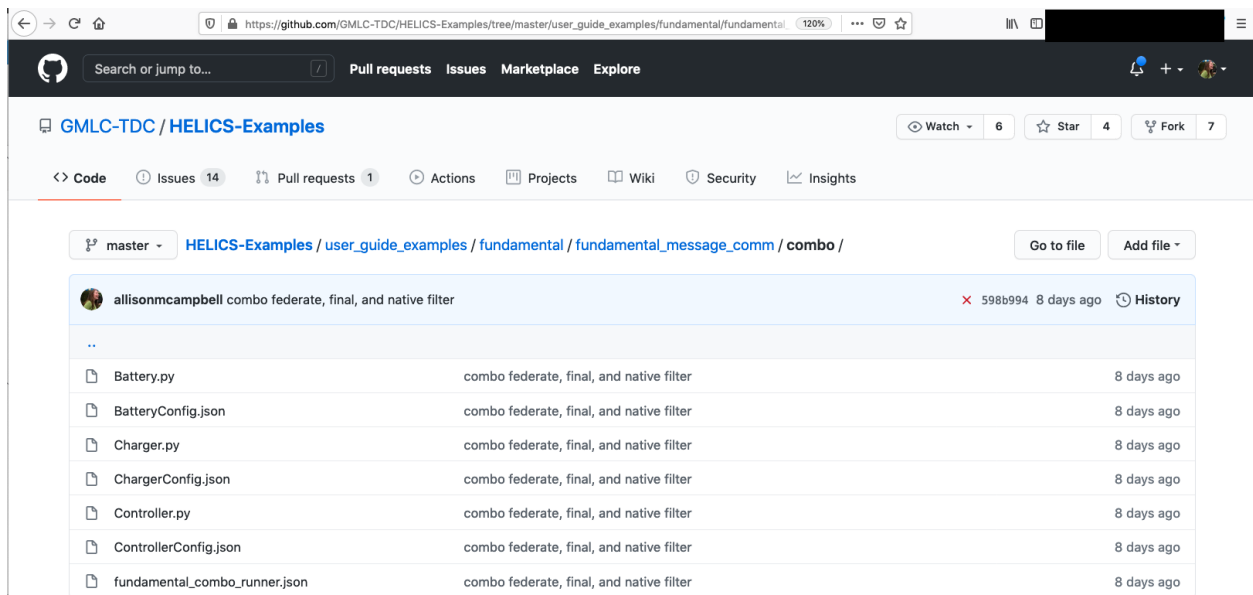
This custom filter federate example expands the *combination federation example* to demonstrate how HELICS filters can be added to endpoints.

This tutorial is organized as follows:

- *Example files*
- *Combination Federates*
 - *When to use pub/subs vs endpoints*
 - *Translation from pub/sub to endpoints*
 - *Co-simulation Execution*
- *Questions and Help*

Example files

All files necessary to run the Federate Integration Example can be found in the [Fundamental examples repository](#):



- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- “runner” JSON to enable `helics_cli` execution of the co-simulation

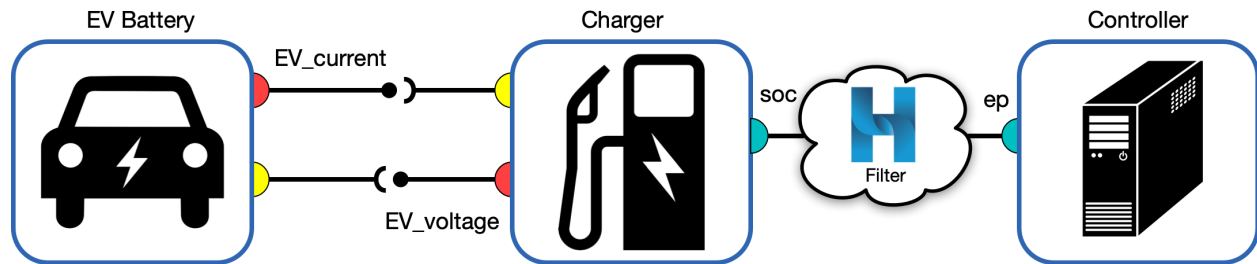
Combination Federates

A quick glance at the [Fundamental examples repository](#) on github will show that almost all these introductory examples are mocked up with two federates. These two federates pass information back and forth, and the examples show different ways this can be done.

This is the only example in the Fundamental series which models three federates – it is also exactly the same model as the *Base Example* in the Advanced series. Why are we introducing a third federate?

In the *Endpoints Example*, we learned how to pass messages between two federates. The problem with this setup – which we will resolve in this example – is that **physical values** should not be modeled with messages/endpoints (see *the example* for a reminder). We introduce a third federate – a **combination federate** – to preserve the handling of physical values among *value federates* and allow for nuanced message passing (and interruption) among *message federates*. The key with combo federates is that they are the go-between for these two types. Combination federates can update (send) values *and* intercept messages. (For a refresher on values and messages, see the section on *Types of Federates*. In brief: values have a physics-based unit, and messages are typically strings).

Here is our new federation of three federates:



We have:

- Battery (**value federate**: passes values with Charger through pub/subs)
- Charger (**combo federate**: passes values with Battery, passes messages with Controller)
- Controller (**message federate**: passes messages with Charger through endpoints)

Redistribution of Federate Roles

The full co-simulation is still asking the same question: “What is the expected instantaneous power draw from a dedicated EV charging garage?” With the introduction of a *Controller* federate, we now have additional flexibility in addressing the nuances to this question. For example, the charging controller does not have direct knowledge of the instantaneous current in the battery – the onboard charger needs to estimate this in order to calculate the EV’s state of charge. Let’s walk through the roles of each federate.

Battery

The Battery federate operates in the same way as in the Base Example. The only difference is that it is now allowed to request a new battery when an existing one is deemed to have a full SOC. This information is in the `charging_voltage` value from the Battery’s subscription to the Charger; if the Charger applies zero voltage, this means the Battery can no longer charge. The Battery federate selects a new battery randomly from three sizes – small, medium, and large – and assigns a random SOC between 0% and 80%.

There are no differences in the config file. As in the Base Example, the Battery federate logs and plots the internally calculated SOC over time at each charging port.

Charger

The Charger federate is now a combination federate – it will communicate via pub/subs with the Battery, and via endpoints with the Controller. This difference from the Base Example Charger is seen in the config file; in addition to the pub/subs with the Battery, there are now also endpoints. Notice that the default destination for each of these named endpoints is the same – there is one controller for all the charging ports.

```
"endpoints": [
  {
    "name": "Charger/EV1.soc",
    "destination": "Controller/ep",
    "global": true
  },
]
```

Since this federate also communicated via endpoints, we need to register them along with the existing pub/subs:

```
##### Registering federate from json #####
fed = h.helicsCreateCombinationFederateFromConfig("ChargerConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
end_count = h.helicsFederateGetEndpointCount(fed)
logger.info(f"\tNumber of endpoints: {end_count}")
sub_count = h.helicsFederateGetInputCount(fed)
logger.info(f"\tNumber of subscriptions: {sub_count}")
pub_count = h.helicsFederateGetPublicationCount(fed)
logger.info(f"\tNumber of publications: {pub_count}")
```

The Charger federate is gaining the new role of *estimating the Battery's current* and shifting the role of *deciding when to stop charging* to the Controller federate.

The Charger federate estimates the Battery federate's current with a new helper function call `estimate_SOC`. The Charger does not know the exact SOC of the Battery; it must estimate the SOC from the effective resistance, which is a function of applied voltage (from the Charger) and the measured current (from the Battery). This is the same function as used in the Battery federate, but with noise added to the measurement of the current.

```
def estimate_SOC(charging_V, charging_A):
    socs = np.array([0, 1])
    effective_R = np.array([8, 150])
    mu = 0
    sigma = 0.2
    noise = np.random.normal(mu, sigma)
    measured_A = charging_A + noise
    measured_R = charging_V / measured_A
    SOC_estimate = np.interp(measured_R, effective_R, socs)

    return SOC_estimate
```

This function is called after the Charger has received the charging current from the Battery federate and needs to update the SOC; if the current is not zero, the Charger estimates the SOC with the inclusion of measurement error on the current. This allows the co-simulation to model the separation of knowledge of the physics between the two federates: the Battery knows its internal current, but the on board Charger must estimate it.

If the current received from the Battery federate is zero, this means that we have plugged a new EV into the charging port and we need to determine the voltage to apply with the Charger. This is accomplished by calling `get_new_EV(1)` and `calc_charging_voltage()`. `get_new_EV(1)` is a helper function which selects the charging level (1, 2, or 3)

based on a set probability distribution and `calc_charging_voltage()` gives the applied voltage for that level. Once a “new EV” (the charging level) has been retrieved, the federate is assigned a SOC of 0 as an initial estimate prior to measuring the current.

The estimated SOC is sent to the Controller every 15 minutes – this mimics an on board charging agent regularly pinging the charging port to confirm if it should continue charging:

```
# Send message to Controller with SOC every 15 minutes
if grantedtime % 900 == 0:
    h.helicsEndpointSendBytesTo(endid[j], "", f"{currentsoc[j]:4f}".encode())
```

The Charger federate is allowed to be interrupted if there is a message from the Controller.

```
# Check for messages from EV Controller
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    instructions = h.helicsMessageGetString(msg)
```

The Charger will receive a message every 15 minutes as well, however it will only change actions if it is told to stop charging. When this happens, the Charger “disengages” from the charging port by applying zero voltage to the Battery.

```
if int(instructions) == 0:
    # Stop charging this EV
    charging_voltage[j] = 0
    logger.info(f"\tEV full; removing charging voltage")
```

Controller

The Controller is a new federate whose role is to decide whether to keep charging an EV based. This decision is based entirely on the estimated SOC calculated by the Charger. Since this decision logic is simple and can be applied to all the EVs modeled by the federation, we can set up the config file with one endpoint:

```
"endpoints": [
    {
        "name": "Controller/ep",
        "global": true
    }
]
```

Note that there is no default destination – the Controller will *respond* to a request for instructions from the Charger. This is accomplished by calling the `h.helicsMessageGetOriginalSource()` API:

```
while h.helicsEndpointHasMessage(endid):

    # Get the SOC from the EV/charging terminal in question
    msg = h.helicsEndpointGetMessage(endid)
    currentsoc = h.helicsMessageGetString(msg)
    source = h.helicsMessageGetOriginalSource(msg)
```

And then sending the message to this source:

```
message = str(instructions)
h.helicsEndpointSendBytesTo(endid, source, message.encode())
```


The Controller federate only operates when it receives a message – it is a *passive* federate. This can be set up by:

1. Initializing the start time of the federate to `h.HELICS_TIME_MAXTIME`:

```
fake_max_time = int(h.HELICS_TIME_MAXTIME)
starttime = fake_max_time
logger.debug(f"Requesting initial time {starttime}")
grantedtime = h.helicsFederateRequestTime(fed, starttime)
```

2. Allow the federate to be interrupted and set a minimum `timedelta` (`ControllerConfig.json`):

```
{
  "name": "Controller",
  ...
  "timedelta": 1,
  "uninterruptible": false,
  ...
}
```

3. Only execute an action when there is a message:

```
while h.helicsEndpointHasMessage(endid):
    pass # placeholder for loop body
```

4. Re-request the `h.HELICS_TIME_MAXTIME` after a message has been received:

```
grantedtime = h.helicsFederateRequestTime(fed, fake_max_time)
```

The message the Controller receives is the SOC estimated by the Charger. If the estimated SOC is greater than 95%, the Controller sends the message back to stop charging.

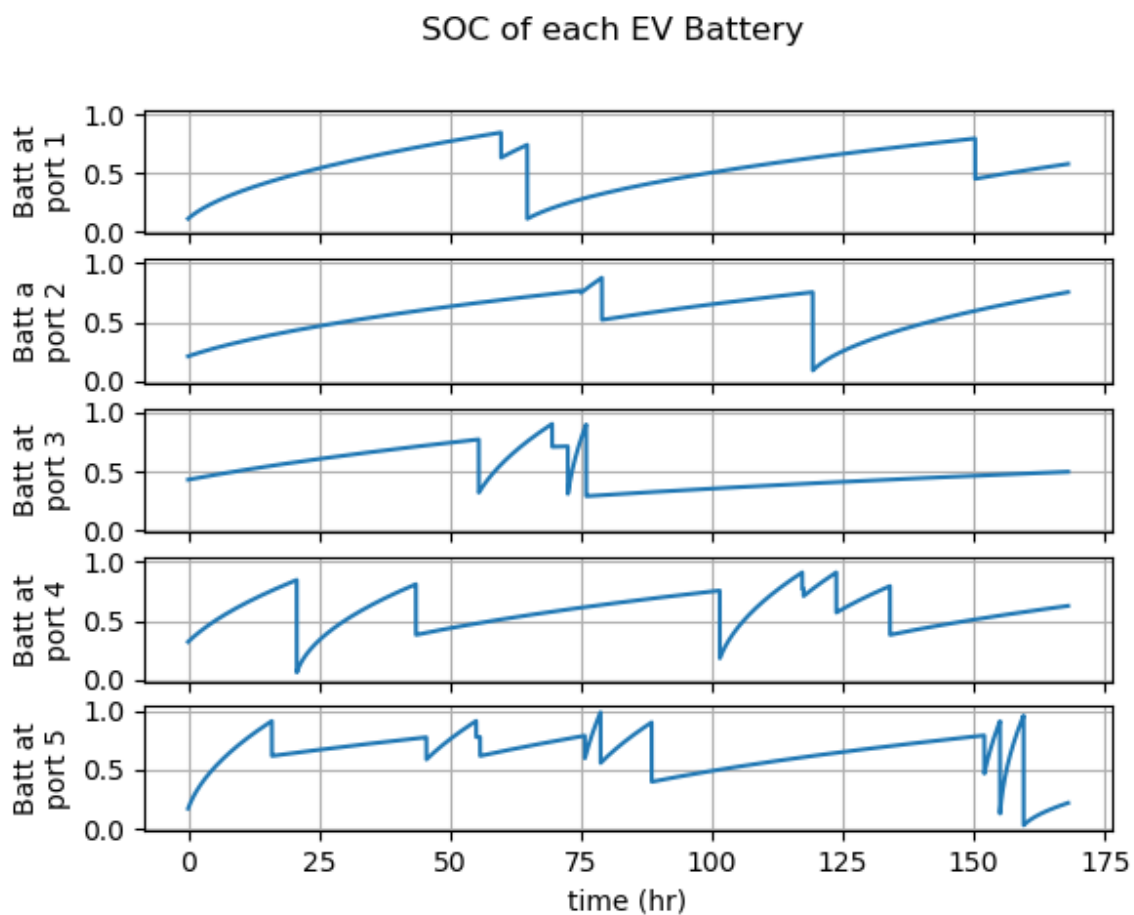
```
soc_full = 0.95
if float(currentsoc) <= soc_full:
    instructions = 1
else:
    instructions = 0
```

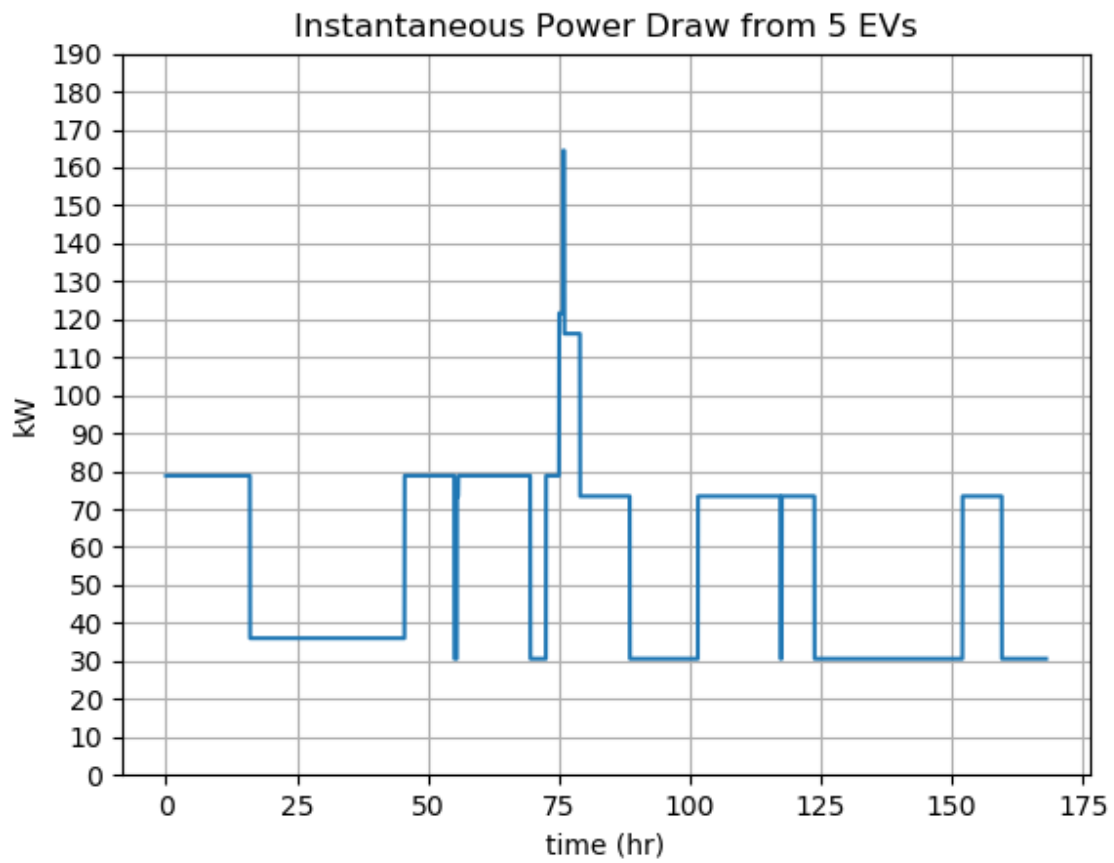
Co-simulation execution

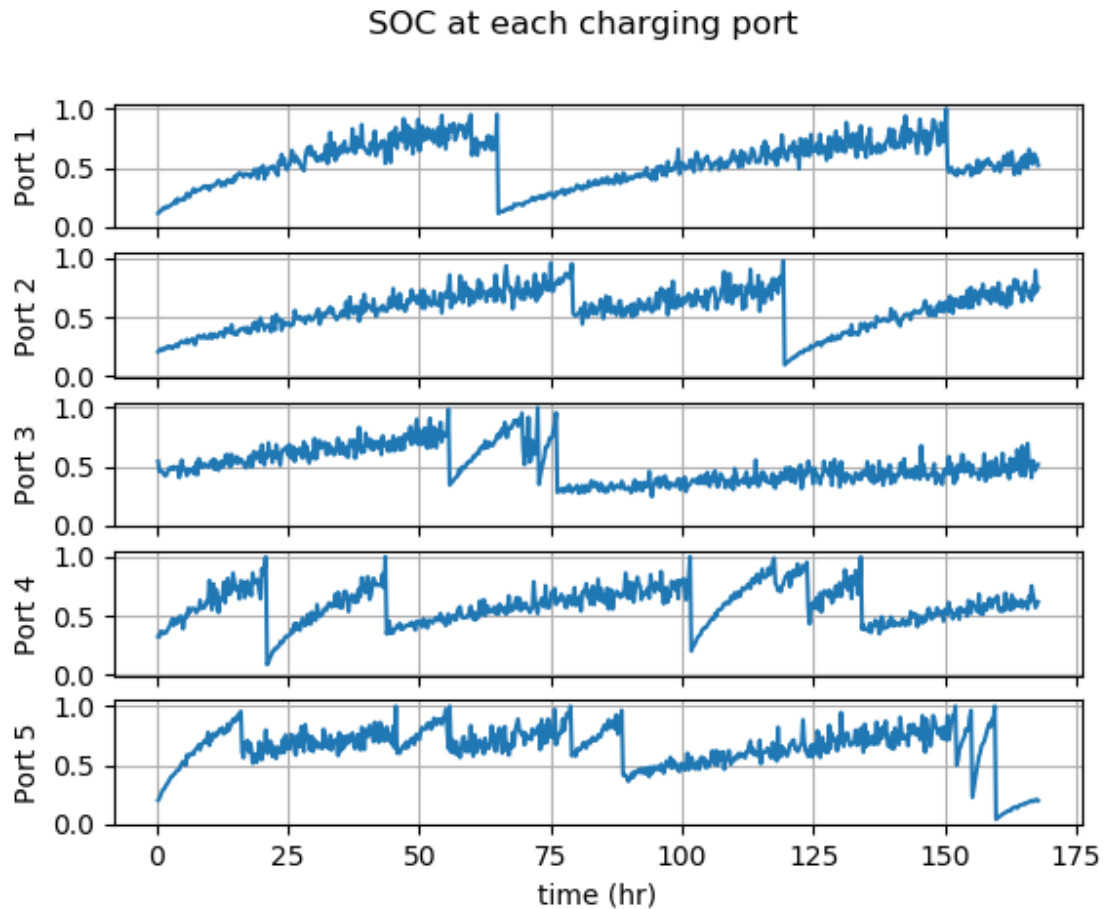
With these three federates – Battery, Charger, and Controller – we have partitioned the roles into the most logical places. Execution of this co-simulation is done as before with `helics_cli`:

```
helics run --path=fundamental_combo_runner.json
```

The resulting figures show the actual on board SOC at each EV charging port, the instantaneous power draw, and the SOC estimated by the on board charger.







Note that we have made a number of simplifying assumptions in this analysis:

- There will always be an EV waiting to be charged (the charging ports are never idle).
- There is a constant number of charging ports – we know what the power draw will look like given a static number of ports, but we do not know the underlying demand for power from EVs.

How would you model an unknown demand for vehicle charging? How would you model idle charging ports? What other simplifications do you see that can be addressed?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Combination Federation with Custom Filter Federates

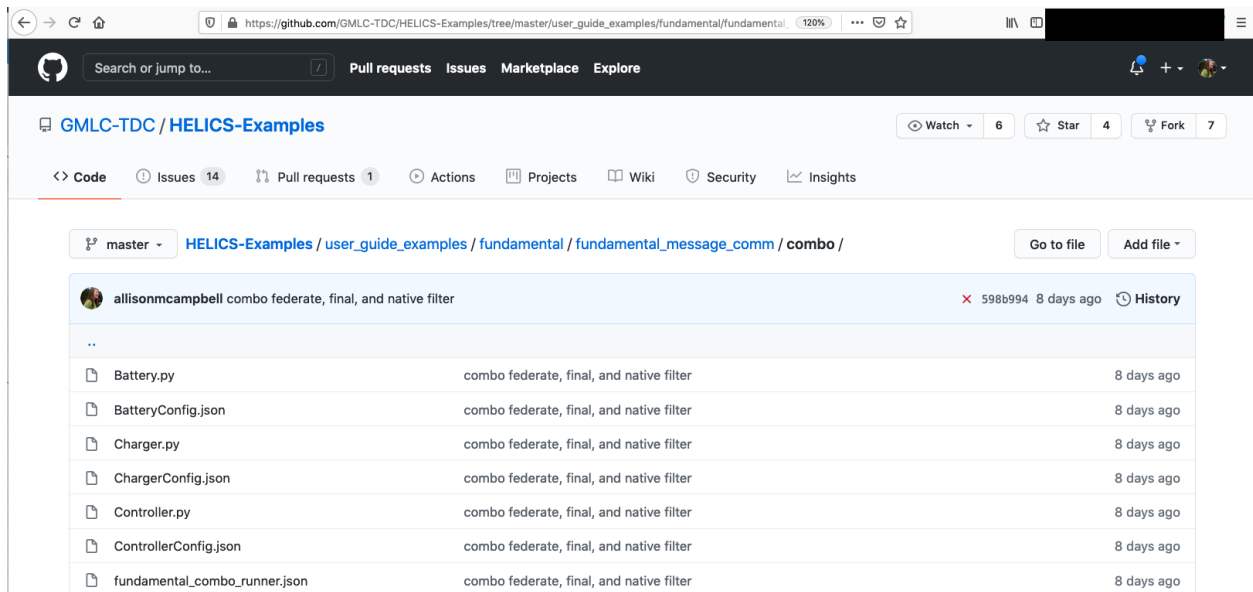
This custom filter federate example expands the Native Filters example to demonstrate . This example assumes the user has already worked through the *Native Filter Example* and understands the role of filters in a HELICS-based co-simulation.

This tutorial is organized as follows:

- *Example files*
- *Filter Federates*
 - *Co-simulation Execution*
- *Questions and Help* talk

Example files

All files necessary to run the Federate Integration Example can be found in the [Fundamental examples repository](#):

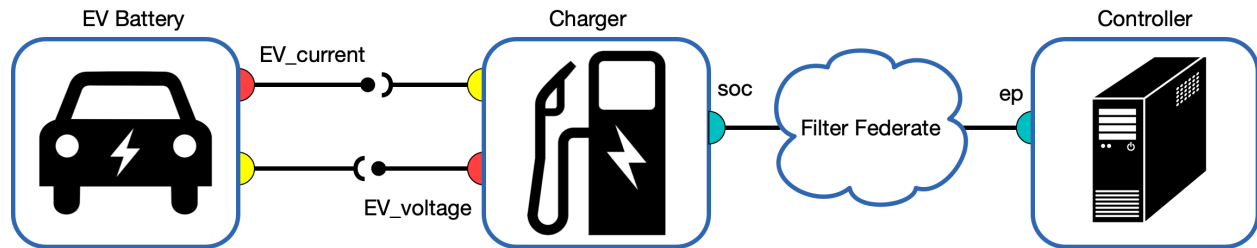


- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- Python program and configuration JSON for Filter federate
- (Bonus Python program where the Filter Federate does no filtering and forwards on all received messages)
- “runner” JSON to enable `helics_cli` execution of the co-simulation

Filter Federate

For situations that require filtering beyond what native HELICS filters can provide, it is possible to create a custom filter federate that acts in specific ways on the messages that receives and forwards on. Not only is it possible to re-create the native helix filter functions such as delay or dropped message but it is also possible act on the payloads of the messages themselves. Some of these functionalities (such as dropped or delayed messages) can be achieved through existing simulation tools such as an [ns-3](#). Even in an ns-3, though custom functionality that would modify message payloads would require writing custom applications in at the industry codebase. Though this example custom filter federate example doesn't implement many of the detailed networking models in an ns-3 such as TCP and IP addressing, it's simplicity is a virtue in that it demonstrates how to implement a custom filter federate to act on HELICS messages an arbitrary manner.

Here is our new federation using a custom filter federate:



We have:

- Battery (**value federate**): passes values with Charger through pub/subs)
- Charger (**combo federate**): passes values with Battery, passes messages with Controller)
- Controller (**message federate**): passes messages with Charger through endpoints)
- Filter (**message federate**): acts on all messages sent between the Charger and the Controller)

Filter Federate Configuration

Though not shown in the Federation diagram, the use of native HELICS filters is an essential part of using a custom filter federate. A "reroute" native helix filter is installed on all the federation endpoints as a part of the configuration of the filter federate. This reroute filter sends all messages from the M points to the filter federate for processing.

```
{
  "name": "Filter",
  "event_triggered": true,
  ...
  "endpoints": [
    {
      "name": "filter/main",
      "global": true
    }
  ],
  "filters": [
    {
      "name": "filterFed",
      "sourcetargets": [
        "Charger/EV1.soc",
        "Charger/EV2.soc",
```

(continues on next page)

(continued from previous page)

```

        "Charger/EV3.soc",
        "Charger/EV4.soc",
        "Charger/EV5.soc",
        "Controller/ep"
    ],
    "operation": "reroute",
    "properties": {
        "name": "newdestination",
        "value": "filter/main"
    }
}
]
}

```

The filter federate will now be granted time whenever a message is sent from any of the existing federation endpoints shown in the `sourcetargets` list. The filter federate has only a single endpoint which it uses to receive the rerouted messages and send on any modified messages.

Additionally, all filter federates should set the `event_triggered` flag as shown above. This increases the timing efficiency managed by HELICS and avoids potential timing lock-ups.

Filter Federate Operations

The filter federate included in this example implements for functions:

- **Random message drops** - A random number generator and a user-adjustable parameter defines what portion of messages traveling through the filter are randomly dropped/deleted.
- **Message delay** - A random number generator and a user-adjustable parameter defines a random delay time for all messages traveling through the filter.
- **Message hacking** - for all messages originating from the controller, a random number generator and a user defined parameter define whether the contents of that message are adjusted. In this case, the contents of the payload are either a one or zero used to indicate where the EV should continue charging; the filter federate simply inverts the message value if it is selected to be hacked.
- **Interference** - Messages that are destined to reach the Controller at the same time can interfere with each other. User-defined parameter defines how closely in time two messages must be arriving to interfere with each other.

Though these operations are just a sample of what any filter federate could do, they are representative of the types of communication system effects that are often sought to be represented through more complex simulators such as ns-3. This simulator contains no network topology; all messages are processed as if traveling through a single communication node. The source code shows the implementation of these functions, and the log files generated by the Filter federate are comprehensive.

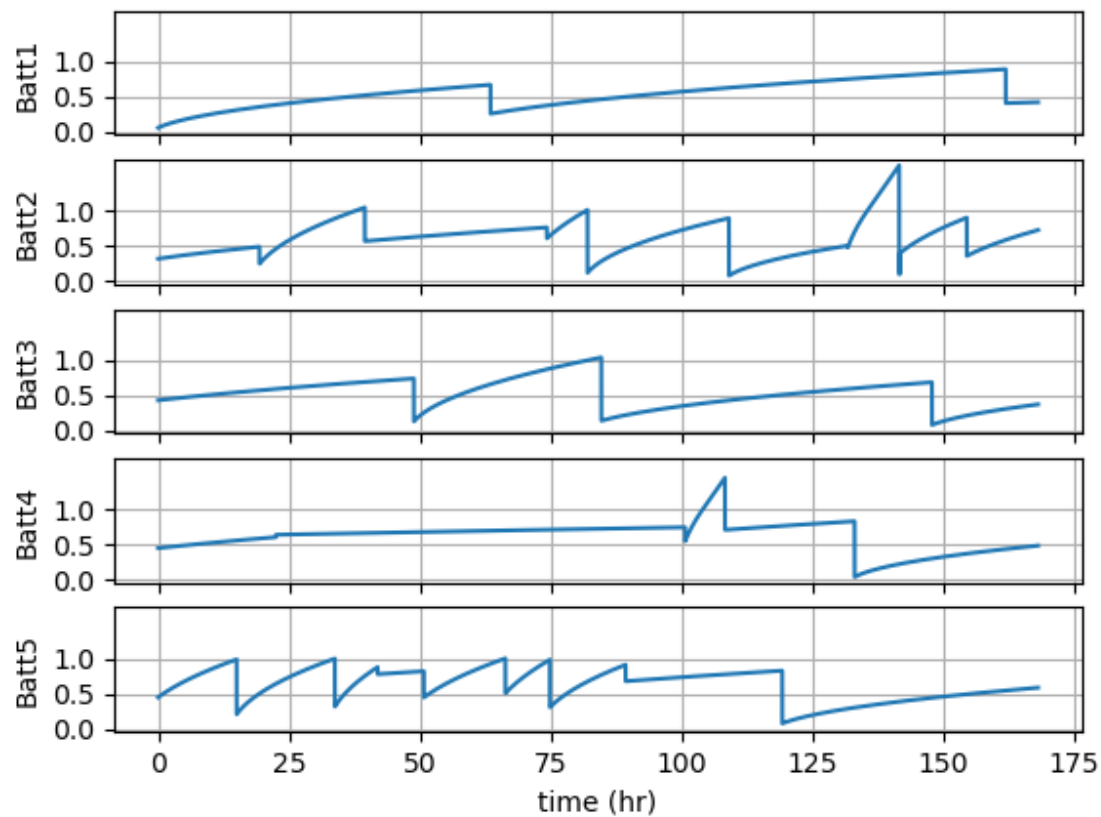
Co-simulation execution

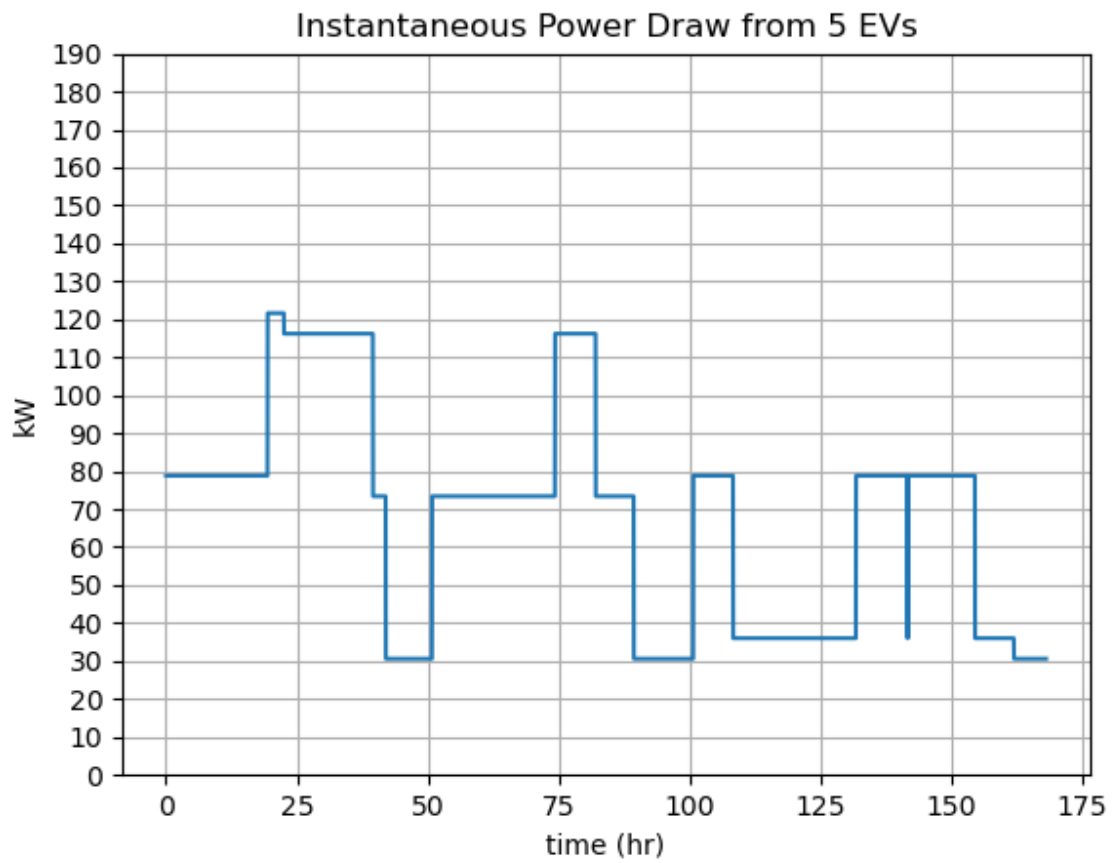
Execution of this co-simulation is done as before with `helics_cli`:

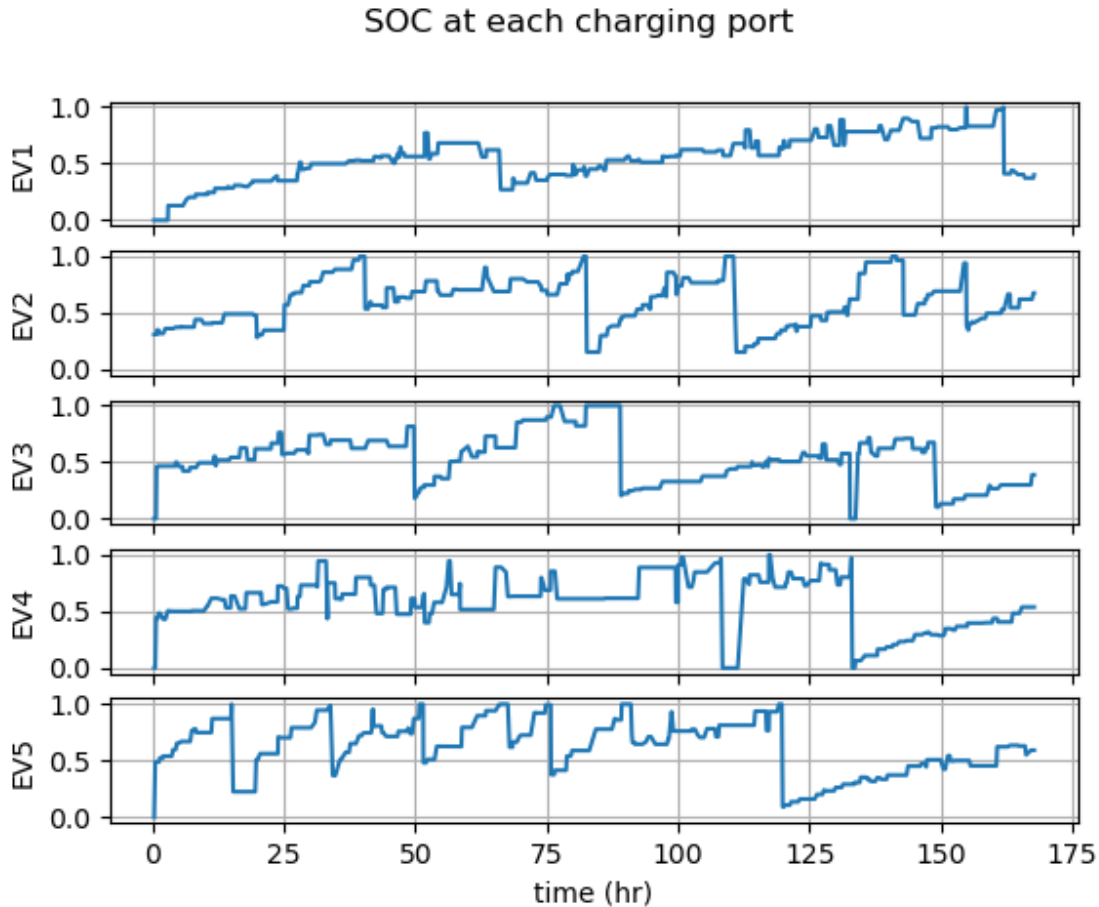
```
helics run --path=./fundamental_filter_runner.json
```

The resulting figures show the actual on board SOC at each EV charging port, the instantaneous power draw, and the SOC estimated by the on board charger.

SOC of each EV Battery







When comparing to the results from the *previous example without any filters*, the effects of the filter federate are clear. By modifying the control signals between the controller and charger it is relatively easy to cause significantly different behavior in the system.

Questions and Help

Do you have questions about HELICS or need help?

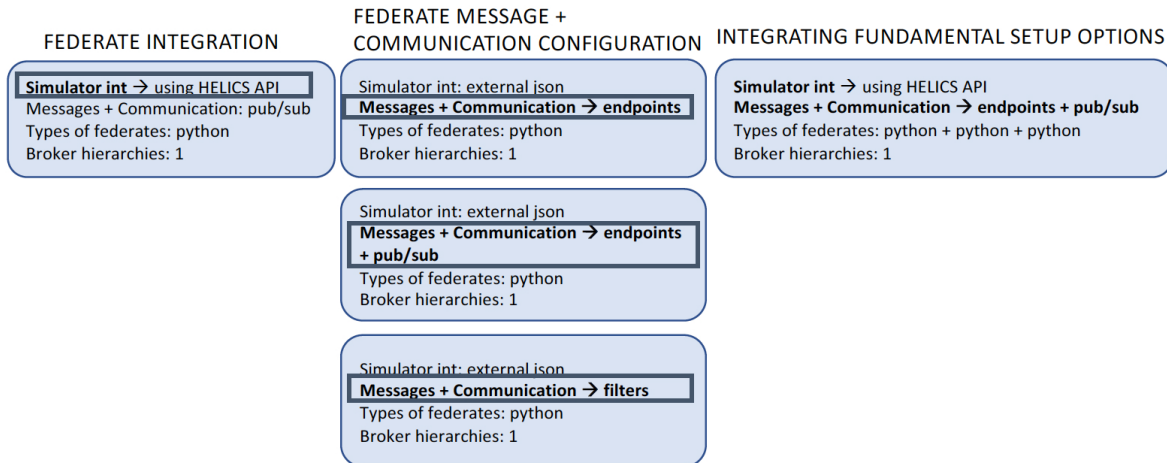
1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

In the *Base Example*, we saw information passed between two federates using publications and subscriptions (pubs/subs). In addition to pubs/subs, where information is passed as *values* (physical parameters of the system), federates can also pass information between **endpoints**, where this information is now a *message*.

This section on message and communication configuration will walk through how to set up two federates to pass messages using *endpoints*, and how to set up three federates which pass between them values and messages with a *combination* of the two configurations. It will also demonstrate how to use native HELICS filters and how to set up a *custom filter federate* to act on messages in-flight between federates.

Default HELICS setup:

Simulator int: external json
 Messages + Communication: pub/sub
 Types of federates: python
 Broker hierarchies: 1

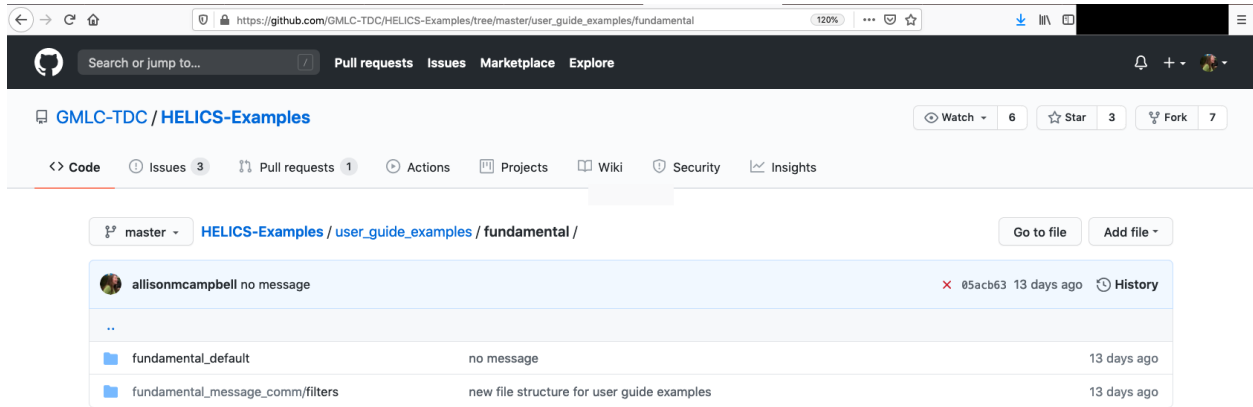
Let's learn about...

The Fundamental examples are meant to build in complexity – if you are new to HELICS, we recommend you start with the Base Example, which is also the recommended default setup. The examples in this section start with the simplest configuration method, which makes assumptions about the system which may not be completely valid but are reasonable for learning purposes.

This page describes the model – what is the research question addressed, and what are the components to a simple HELICS co-simulation:

- *Where is the code?*
- *What is this Co-simulation doing?*
- *HELICS Components*
 - *Register and Configure Federates*
 - *Enter Execution Mode*
 - *Define Time Variables*
 - *Initiate Time Steps for the Time Loop*
 - *Send/Receive Communication between Federates*
 - *Finalize Co-simulation*

The code for the [Fundamental examples](#) can be found in the HELICS-Examples repository on github. If you have issues navigating the examples, visit the [HELICS Gitter page](#) or the [user forum on GitHub](#).



The Fundamental Examples model the interaction between five electric vehicles (EVs) each connected to a charging station – five EVs, five charging stations. You can imagine that five EVs enter a parking garage filled with charging stations (charging garage). We will be modeling the EVs as if they are solely their on-board batteries.

Imagine you are the engineer assigned to assess how much power will be needed to serve EVs in this charging garage. The goal of the co-simulation is to calculate the **instantaneous power draw** from the EVs in the garage.

Some questions you might ask yourself include:

1. How many EVs are likely to be charging at any given time?
2. What is the charge rate (power draw limitation) for these EVs?
3. What is the battery size (total capacity) for these EVs?

For now, we've decided that there will always be five EVs and five charging stations. We can also define a few functions to assign the charge rates and the battery sizes. This requires some thinking about *which* federate will manage *what* information.

The co-simulation has two federates: one for the EVs, and one for the Chargers. The batteries on board the EVs will remain the domain of the EVs and inform the EV state of charge (SOC). The Chargers will manage the voltage applied to the EV batteries, and will retain knowledge of the rate of charge.

Within the federate `Battery.py`, EV battery sizes can be generated using the function `get_new_battery`:

```
def get_new_battery(numBattery):
    # Probabilities of a new EV battery having small capacity (lv11),
    # medium capacity (lv12), and large capacity (lv13).
    lv11 = 0.2
    lv12 = 0.2
    lv13 = 0.6

    # Batteries have different sizes:
    # [25,62,100]
    listOfBatts = np.random.choice(
        [25, 62, 100], numBattery, p=[lv11, lv12, lv13]
    ).tolist()
    return listOfBatts
```

Within the federate `Charger.py`, the charge rate for each EV is generated using the function `get_new_EV`:

```
def get_new_EV(numEVs):
    # Probabilities of a new EV charging at the specified level.
    lv11 = 0.05
```

(continues on next page)

(continued from previous page)

```

lvl2 = 0.6
lvl3 = 0.35
listOfEVs = np.random.choice([1, 2, 3], numEVs, p=[lvl1, lvl2, lvl3]).tolist()
numLvl1 = listOfEVs.count(1)
numLvl2 = listOfEVs.count(2)
numLvl3 = listOfEVs.count(3)

return numLvl1, numLvl2, numLvl3, listOfEVs

```

The probabilities assigned to each of these functions are placeholders – a more advanced application can be found in the [Orchestration Tutorial](#).

Now that we know these three quantities – the number of EVs, the capacity of their batteries, and their charge rates, we can build a co-simulation from the two federates. The `Battery.py` federate will update the SOC of each EV after it receives the voltage from the `Charger.py` federate. The `Charger.py` federate will send a voltage signal to the EV until it tells the Charger it has reached its full capacity.

The `Battery.py` federate can tell us the SOC of each EV throughout the co-simulation, and the `Charger.py` federate can tell us the aggregate power draw from all the EVs throughout the co-simulation. The co-simulation will be run for one week.

We know conceptually what we want to (co-)simulate. What are the necessary HELICS components to knit these two federates into one co-simulation?

The first task is to register and configure the federates with HELICS within each python program:

```

##### Registering federate and configuring from JSON#####
fed = h.helicsCreateValueFederateFromConfig("BatteryConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
print(f"Created federate {federate_name}")

```

Since we are configuring with external JSON files, this is done in one line!

The HELICS co-simulation starts by instructing each federate to enter execution mode.

```

##### Entering Execution Mode #####
h.helicsFederateEnterExecutingMode(fed)
logger.info("Entered HELICS execution mode")

```

Time management is a vital component to HELICS co-simulations. Every HELICS co-simulation needs to be provided information about the start time (`grantedtime`), the end time (`total_interval`) and the time step (`update_interval`). Federates can *step through time at different rates*, and it is allowable to have federates start and stop at different times, but this must be curated to meet the needs of the research question.

```

hours = 24 * 7
total_interval = int(60 * 60 * hours)
update_interval = int(
    h.helicsFederateGetTimeProperty(fed, h.helics_property_time_period)
)
grantedtime = 0

```

Starting the co-simulation time sequence is also a function of the needs of the research question. In the Base Example, the EVs will already be “connected” to the Chargers and will be waiting for the voltage signal from the Charger. This means we need to set up a signal to send from the Charger to the EV *before* the EV requests the signal.

In the `Battery.py` federate, Time is initiated by starting a while loop and requesting the first time stamp:

```
while grantedtime < total_interval:

    # Time request for the next physical interval to be simulated
    requested_time = grantedtime + update_interval
    logger.debug(f"Requesting time {requested_time}")
    grantedtime = h.helicsFederateRequestTime(fed, requested_time)
    logger.debug(f"Granted time {grantedtime}")
```

In the `Charger.py` federate, we need to send the first signal **before** entering the time while loop. This is accomplished by requesting an initial time (outside the while loop), sending the signal, and then starting the time while loop:

```
# Blocking call for a time request at simulation time 0
initial_time = 60
logger.debug(f"Requesting initial time {initial_time}")
grantedtime = h.helicsFederateRequestTime(fed, initial_time)
logger.debug(f"Granted time {grantedtime}")

# Apply initial charging voltage
for j in range(0, pub_count):
    h.helicsPublicationPublishDouble(pubid[j], charging_voltage[j])
    logger.debug(
        f"\tPublishing charging voltage of {charging_voltage[j]} "
        f" at time {grantedtime}"
    )

##### Main co-simulation loop #####
# As long as granted time is in the time range to be simulated...
while grantedtime < total_interval:

    # Time request for the next physical interval to be simulated
    requested_time = grantedtime + update_interval
    logger.debug(f"Requesting time {requested_time}")
    grantedtime = h.helicsFederateRequestTime(fed, requested_time)
    logger.debug(f"Granted time {grantedtime}")
```

Once inside the time loop, information is requested and sent between federates at each time step. In the Base Example, the federates first request information from the handles to which they have subscribed, and then send information from the handles from which they publish.

The `Battery.py` federate first asks for voltage information from the handles to which it subscribes:

```
# Get the applied charging voltage from the EV
charging_voltage = h.helicsInputGetDouble((subid[0]))
logger.debug(
    f"\tReceived voltage {charging_voltage:.2f} from input"
    f" {h.helicsSubscriptionGetKey(subid[0])}"
)
```

And then (after doing some internal calculations) publishes the charging current of the battery at its publication handle:

```
# Publish out charging current
h.helicsPublicationPublishDouble(pubid[j], charging_current)
logger.debug(f"\tPublished {pub_name[j]} with value " f"{charging_current:.2f}")
```

Meanwhile, the `Charger.py` federate asks for charging current from the handles to which it subscribes:

```
charging_current[j] = h.helicsInputGetDouble((subid[j]))
logger.debug(
    f"\tCharging current: {charging_current[j]:.2f} from "
    f"input {h.helicsSubscriptionGetKey(subid[j])}"
)
```

And publishes the charging voltage at its publication handle:

```
# Publish updated charging voltage
h.helicsPublicationPublishDouble(pubid[j], charging_voltage[j])
logger.debug(
    f"\tPublishing charging voltage of {charging_voltage[j]} " f" at time {grantedtime}"
)
```

After all the time steps have completed, it's good practice to finalize the co-simulation by freeing the federates and closing the HELICS libraries:

```
status = h.helicsFederateFinalize(fed)
h.helicsFederateFree(fed)
h.helicsCloseLibrary()
```

1.4.2 Advanced Examples

Default Advanced Example

The Advanced Base example walks through a HELICS co-simulation between three python federates, one of each type: value federate (`Battery.py`), message federate (`Controller.py`), and combination federate (`Charger.py`). This serves as the starting point for many of the other advanced examples and is an extension of the [Base Example](#).

Default HELICS setup:

Simulator int: external json
 Messages + Communication: endpoints
 + pub/sub
 Types of federates: python
 Broker hierarchies: 1

The Advanced Base Example tutorial is organized as follows:

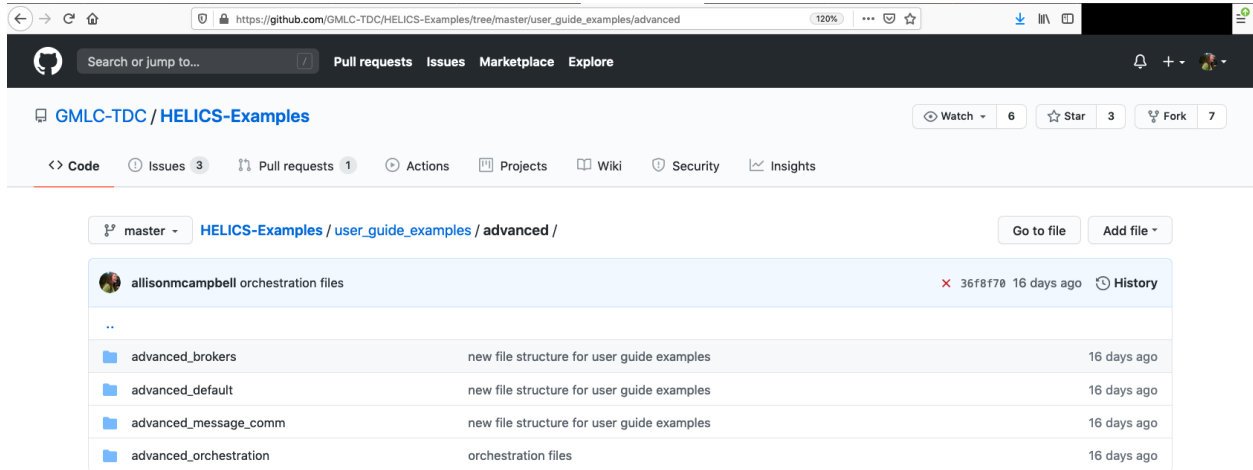
- *Example files*
- *Co-simulation Setup*
 - *Messages + Values*

– *Co-simulation Execution and Results*

- *Questions and Help*

Example files

All files necessary to run the Advanced Base Example can be found in the [Advanced Examples repository](#):



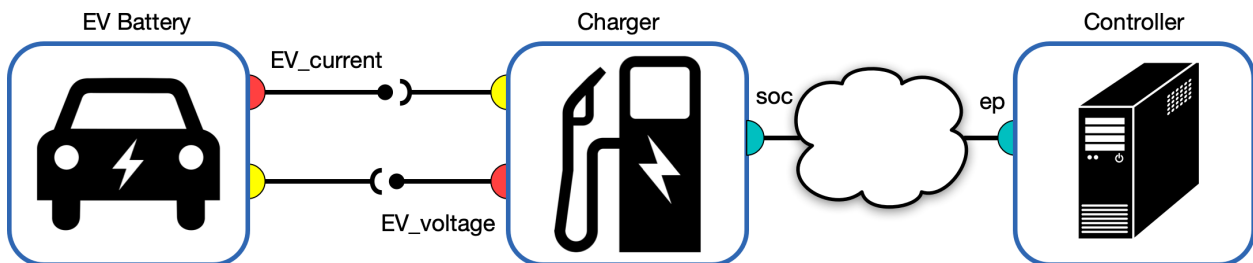
The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- “runner” JSON to enable `helics_cli` execution of the co-simulation

Co-simulation Setup

Messages + Values

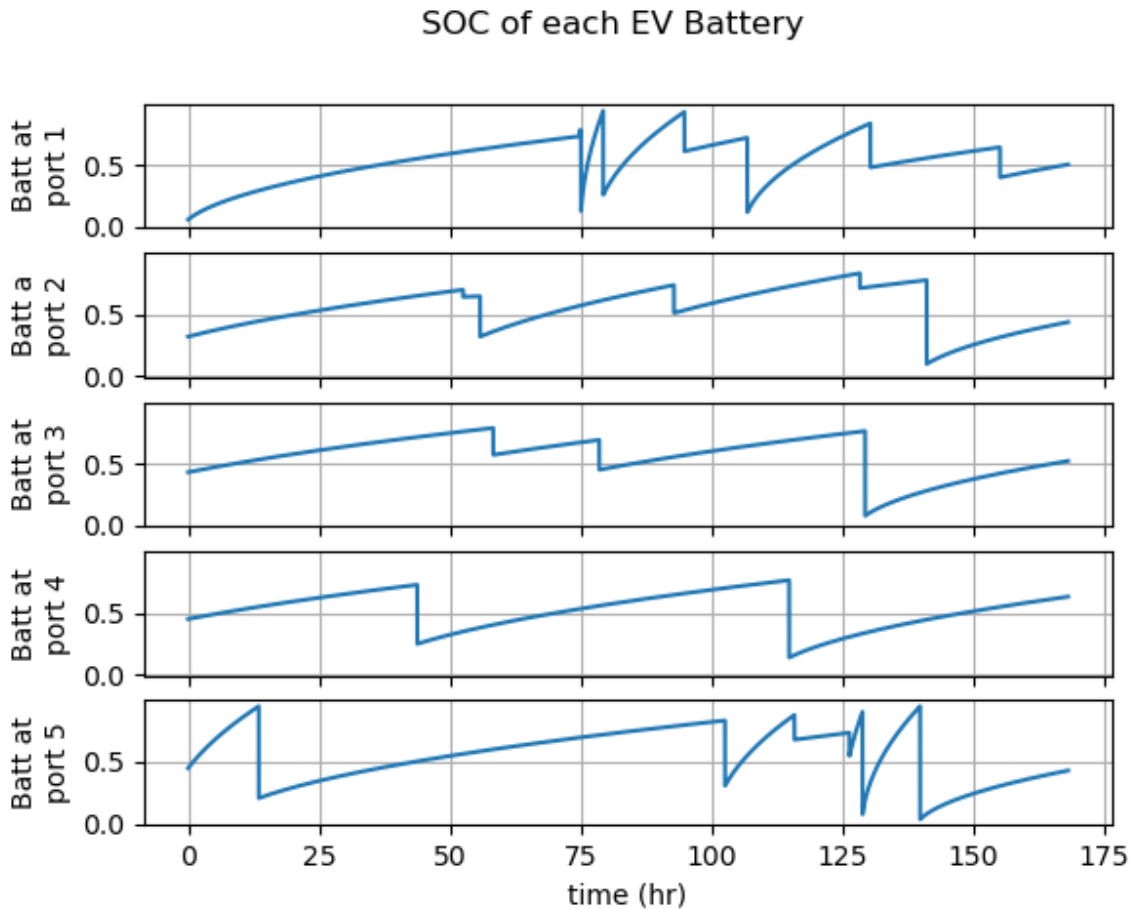
As you may or may not have read in the User Guide, one of the key differences between value exchange and the message exchange is that value exchange paths are defined once the federation has been initialized but message exchanges are dynamic and can travel from any endpoint to any endpoint throughout the co-simulation. The diagram below shows the three federates used in this example with the representative handles for both the value and message exchanges.



Co-simulation Execution and Results

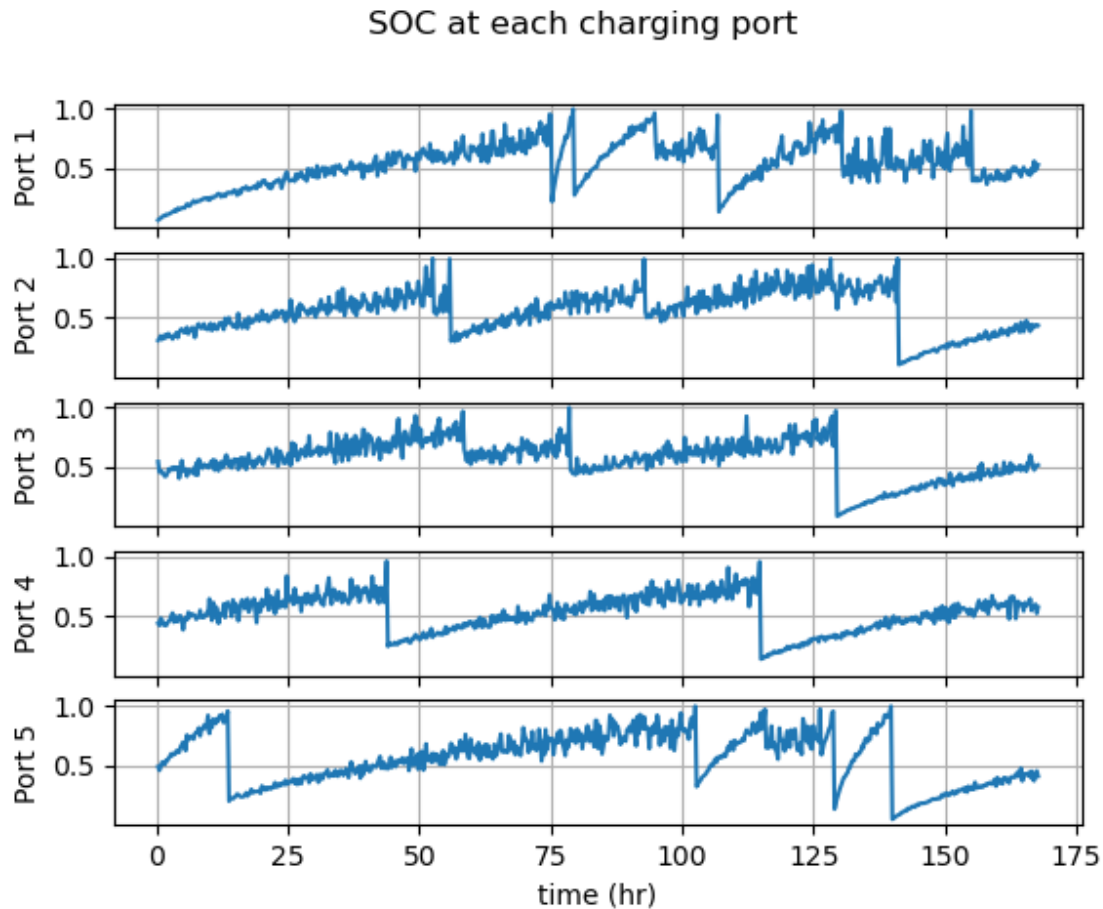
As in the *Fundamental Base Example*, `helics_cli` is used to launch the co-simulation:

```
> helics run --path=advanced_default_runner.json
```



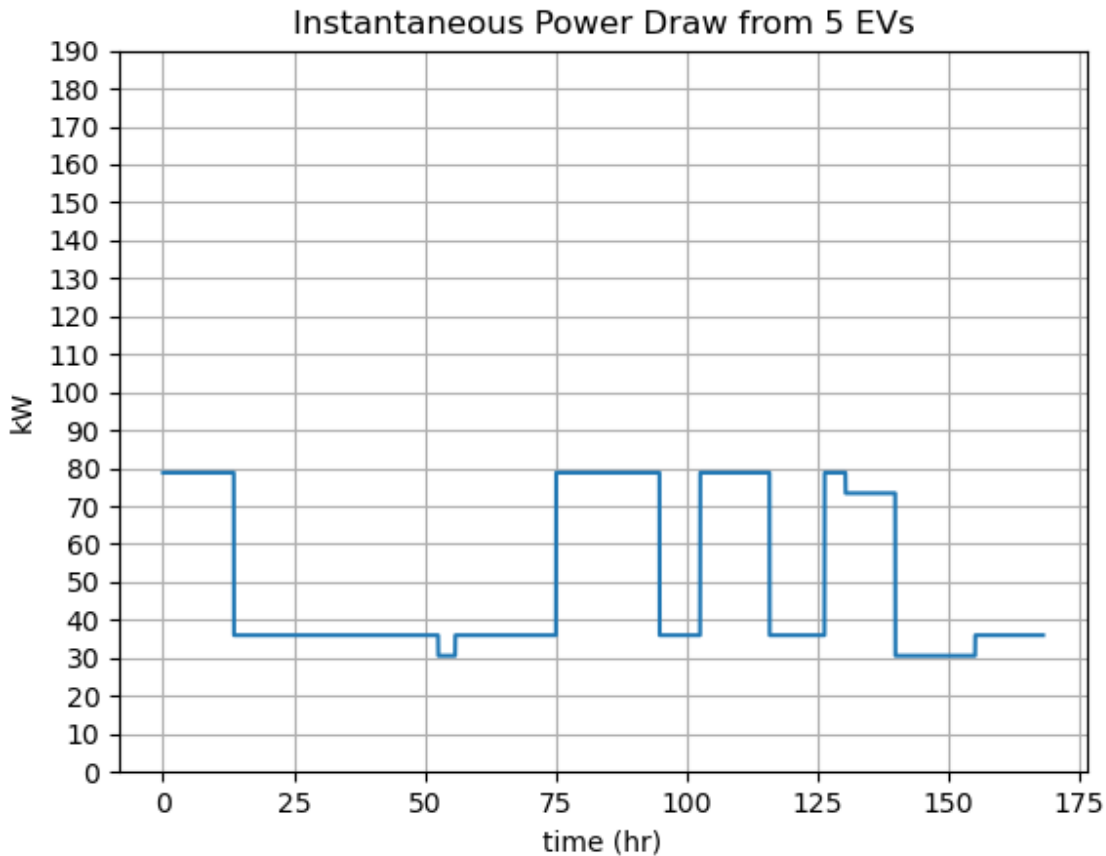
This is the view of each battery as it is charged and two things are immediately obvious:

1. The impact of the charging level is pronounced. The first Batt1 takes almost half the simulation to charge but when its replacement is placed on the charger, it starts at a similar SOC but charges in a fraction of the time. The impact of the charging power supported by each EV is significant.
2. Most of the batteries fail to reach 100% SOC, some dramatically so. This is due to the current measurement error leading to a mis-estimate of SOC and thus premature termination of the charging. This can be seen the following graph



As previously mentioned, the current measurement noise is a function of the total magnitude of the current and thus as the battery charges up and the current draw drops, the noise in the measurement becomes a bigger fraction of the overall value. This results in the noisiest SOC estimates at higher SOC values. This is clearly seen in the EV1 value that starts the co-simulation relatively smooth and steadily increases in noisiness.

This graph also clearly shows that each EV was estimated to have a 100% SOC when the charging was terminated even though we know from the previous graph that full charge had not been reached.



The data shown in the power graph is arguably the point of the analysis. It shows our maximum charging power for this simulated time as 80 kW. If this is the only simulation result we have, we would be inclined to use this as a design value for our electricity delivery infrastructure. More nuanced views could be had, though, by:

1. Running this co-simulation multiple times using a different random seed to see if 80 kW is truly the maximum power draw. (We do a version of this in an [example demonstrating how to run multiple HELICS co-simulations simultaneously](#) on a single compute node.)
2. Plotting the charging power as a histogram to get a better understanding of the distribution of the instantaneous charging power. (We also do this as part of our [example on using an orchestration tool to use HELICS co-simulations as part of a more complex analysis](#).)

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Brokers - Simultaneous Co-simulations

Default HELICS setup:

```

Simulator int: external json
Messages + Communication: endpoints
+ pub/sub
Types of federates: python
Broker hierarchies: 1
  
```

Let's learn about...

FEDERATE MESSAGE + COMMUNICATION CONFIGURATION

```

Simulator int: external json
Messages + Communications →
pub/sub + multi-source inputs
Types of federates: python
Brokers: 1
  
```

```

Simulator int: external json
Messages + Communication →
endpoints + queries
Types of federates: python
Brokers: 1
  
```

```

Simulator int: external json
Messages + Communication →
python ns3
Types of federates → python
python
Brokers: 1
  
```

```

Simulator int: external json
Messages + Communication →
pub/sub + ns3
Types of federates → python + ns3
Brokers: 1
  
```

BROKERS

```

Simulator int: external json
Messages + Communication:
pub/sub
Types of federates: python
Brokers → multiple core types
  
```

```

Simulator int: external json
Messages + Communication:
pub/sub
Types of federates: python
Brokers → hierarchies
  
```

ORCHESTRATION

```

Simulator int: external json
Messages + Communication:
pub/sub
Types of federates: python
Brokers → MERLIN for
orchestration
  
```

MANY TYPES OF FEDERATES

```


Simulator int → external JSON
Messages + Communication: pub/sub
Types of federates → python + GLD
Broker hierarchies: 1
  
```




This example shows how to configure a HELICS co-simulation so that multiple co-simulations can run simultaneously on one computer. Understanding this configuration is a pre-requisite to running the other advanced broker examples (which also involve multiple brokers running on one computer).

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Fundamental Examples*
 - * *HELICS Differences*
 - * *Research Question Complexity Differences*
- *Execution and Results*

Where is the code?


The code for the [Advanced examples](#) can be found in the HELICS-Examples repository on GitHub. This example on [simultaneous co-simulations](#) can be found [here](#). If you have issues navigating the examples, visit the [HELICS Gitter page](#) or the [user forum](#) on GitHub.












[GMLC-TDC / HELICS-Examples](#)

 Watch 6
 Star 4
 Fork 7

[Code](#)
[Issues 14](#)
[Pull requests 1](#)
[Actions](#)
[Projects](#)
[Wiki](#)
[Security](#)
[Insights](#)

master
[HELICS-Examples / user_guide_examples / advanced / advanced_brokers / hierarchies /](#)
[Go to file](#)
[Add file](#)
...


trevorhardy Remove unnecessary python parameter in script call
 4e5bca9 on Nov 17, 2020
[History](#)

..		
 Battery.py	Update output graph file names	2 months ago
 BatteryConfig.json	Add working broker hierarchy example	2 months ago
 Charger.py	Update output graph file names	2 months ago
 ChargerConfig.json	Add working broker hierarchy example	2 months ago
 Controller.py	Update output graph file names	2 months ago
 ControllerConfig.json	Add working broker hierarchy example	2 months ago
 broker_hierarchy_example.md	Add working broker hierarchy example	2 months ago
 broker_hierarchy_runner_A.json	Remove federate value from root broker `exec` string	2 months ago
 broker_hierarchy_runner_B.json	Remove unnecessary python parameter in script call	2 months ago
 broker_hierarchy_runner_C.json	Add working broker hierarchy example	2 months ago

What is this co-simulation doing?

Using the exact same federates as in the *Advanced Default example*, the same co-simulation is run multiple times (simultaneously) with different random number generator seeds. The example both demonstrates how to run multiple HELICS co-simulations simultaneously on one computer without the messages between federates getting mixed up, but also shows a simple way to do sensitivity analysis. A better way is shown later in the *orchestration example*.

Differences compared to the advanced default example

Two primary changes:

1. This example contains a set of co-simulations with each instance using a different random number generator seed in Battery.py

```
if __name__ == "__main__":
    np.random.seed(2608)
```

and Charger.py

```
if __name__ == "__main__":
    np.random.seed(1490)
```

The values shown above are from `federation_1`. Identical lines with alternative values can be found in `federation_2` and `federation_3`.

1. The brokers are configured to ensure that messages from one federation do not get routed to federates in another federation.

HELICS differences

With no extra configuration, it is only possible to run one HELICS co-simulation on a given computer. In the most popular HELICS cores (ZMQ being the most common, by far), messages are sent between federates using the networking stack. (There are other ways, though. For example, the IPC core uses the Boost library inter-process communication.) If you want to run multiple co-simulations on one compute need, an extra step needs to be taken to keep the messages from each federation separate from each other and non-interfering. Since we're using the network stack, this can be easily accomplished by assigning each broker a unique port to use. Looking at the federation launch config files, you can see this clearly expressed:

federation_1_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20100",
      "host": "localhost",
      "name": "broker"
    },
  ],
}
```

federation_2_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20200",
      "host": "localhost",
      "name": "broker"
    },
  ],
}
```

federation_3_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20300",
      "host": "localhost",
      "name": "broker"
    },
  ],
}
```

Research question complexity differences

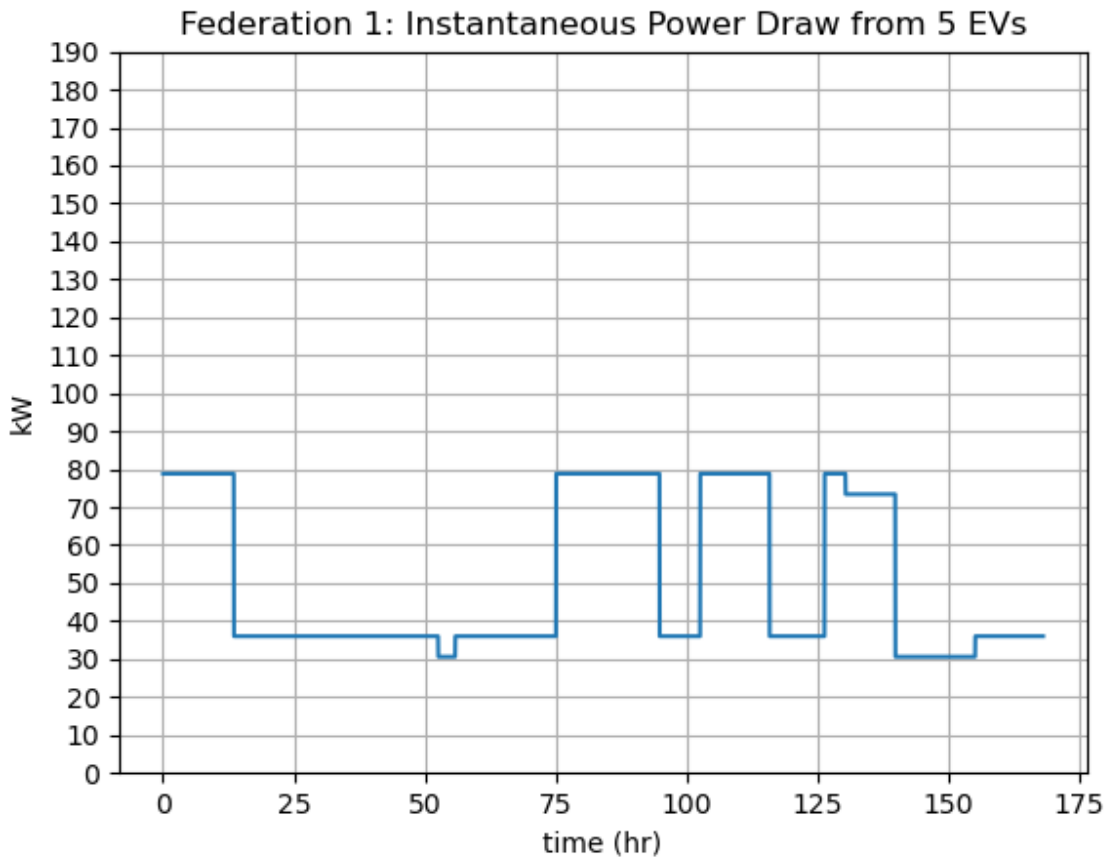
The Advanced Default example uses a random number generator to determine things like initial state-of-charge of the battery and charging power of the individual EV batteries. These factors have an impact on the charging duration for each EV battery and the peak charging power seen over the duration of the simulation. Since the later is *the* key metric of the simulation experiment, there is strong motivation to vary the seed value for the random number generator to expand the range of results, effectively increasing the sample size. These co-simulations could each be run serially but assuming the computer in question has the horsepower, there's no reason not to run them in parallel.

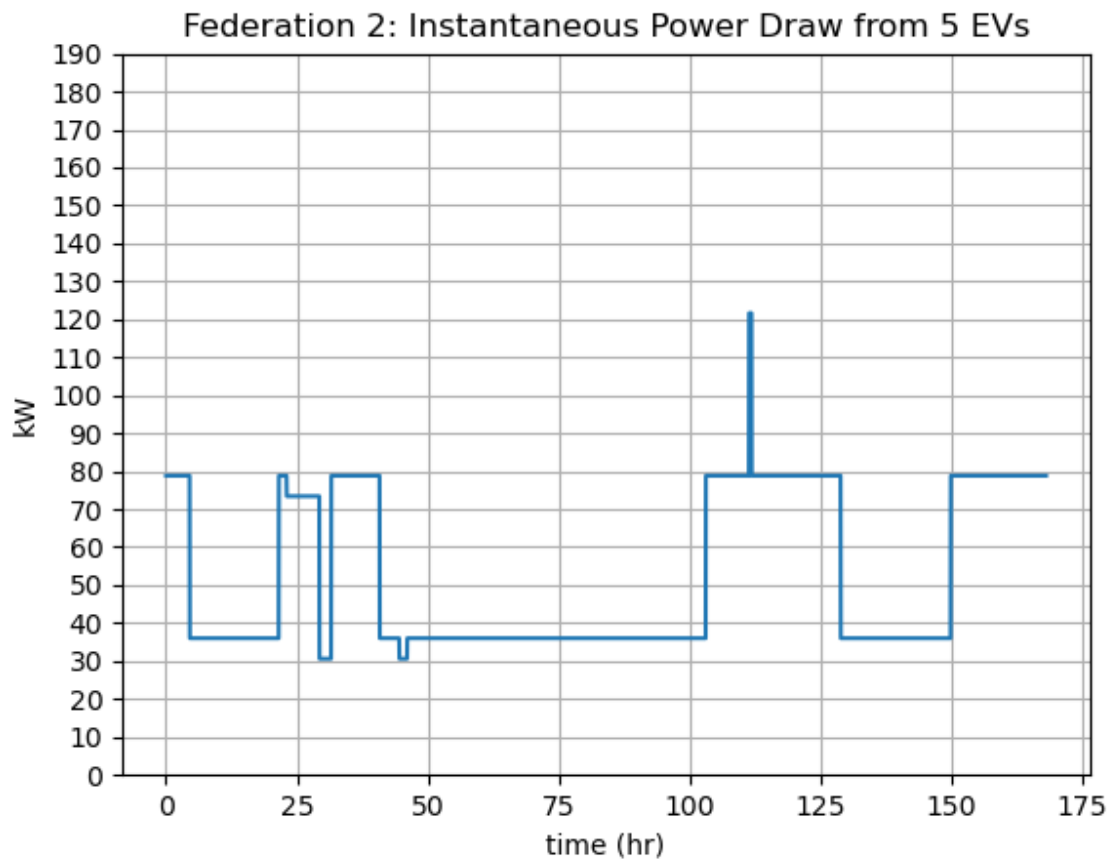
Execution and Results

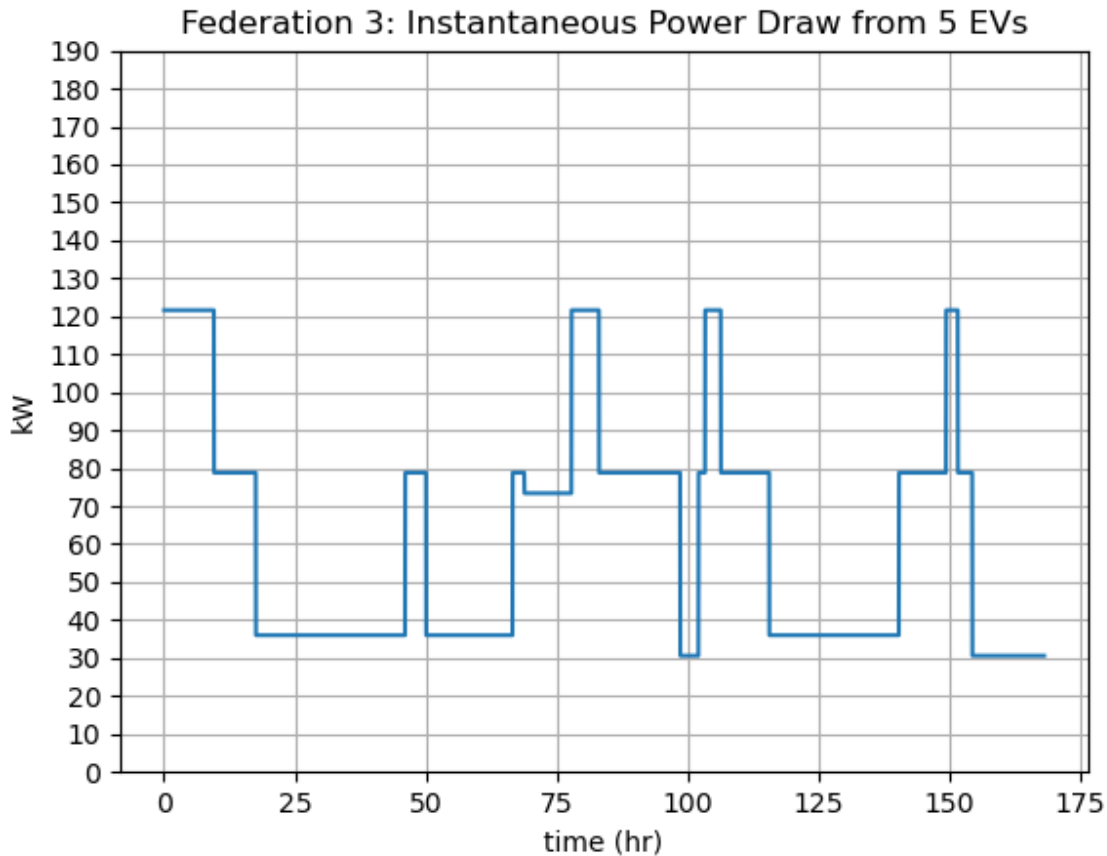
To run the co-simulations simultaneously, all that is required is having `helics_cli` launch each individually. The trailing `&` in the shell commands below background the command and return another shell prompt to the user.

```
$ helics run --path=./federation_1/federation_1_runner.json &      $ helics run --path=.
/federation_2/federation_2_runner.json &                          $ helics run --path=./federation_2/
federation_3_runner.json &
```

The peak charging results are shown below. As can be seen, the peak power amplitude and the total time at peak power are impacted by the random number generator seed.







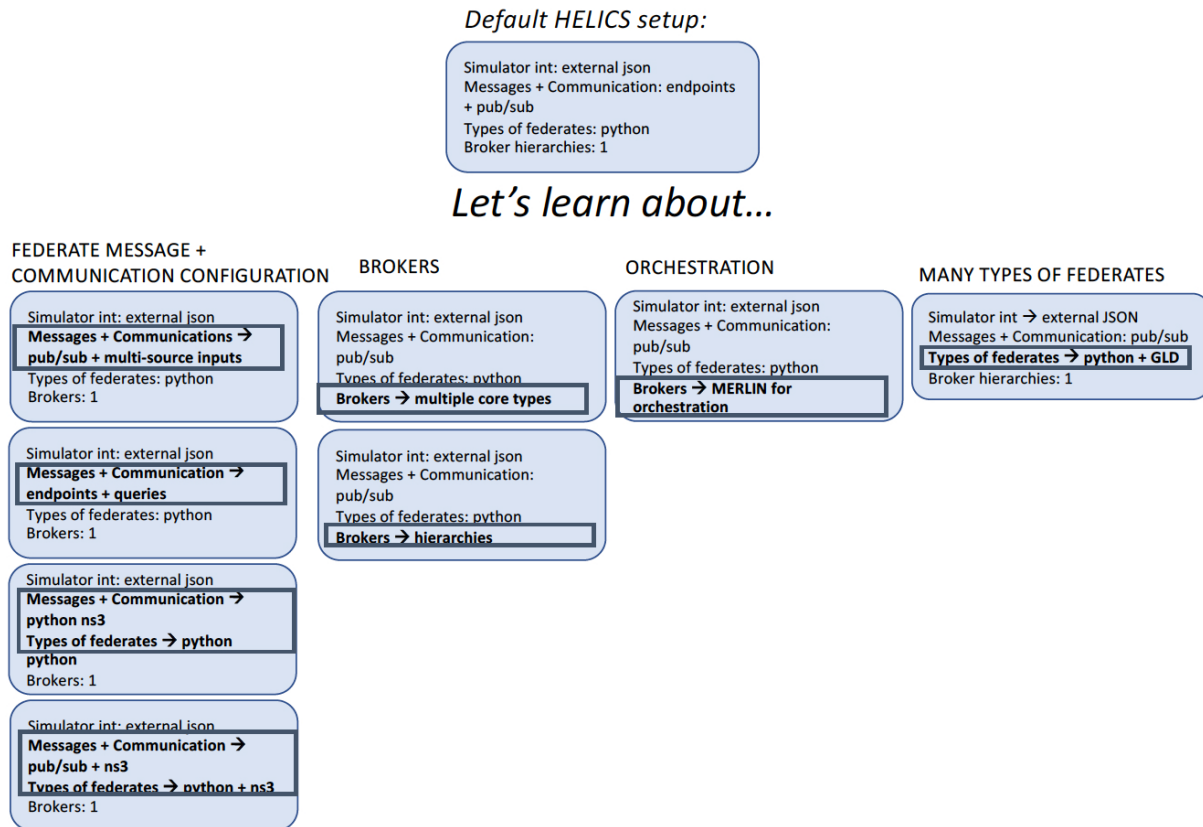
To do a more legitimate sensitivity analysis to the population of EVs that are being charged, a sample size larger than three is almost certainly necessary. *We've put together another example* to show how to orchestrate running larger sets of co-simulations to address exactly these kinds of needs.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Brokers - Hierarchies



This example shows how to configure a HELICS co-simulation to allow the use of multiple brokers in a single co-simulation.

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Advanced Default Example*
 - * *HELICS Differences*
 - *HELICS Components*
- *Execution and Results*

Where is the code?

The code for the [Advanced examples](#) can be found in the HELICS-Examples repository on GitHub. This example on [broker hierarchies](#) can be found [here](#). If you have issues navigating the examples, visit the [HELICS Gitter page](#) or the [user forum](#) on GitHub.

[GMLC-TDC / HELICS-Examples](#)

Watch 6
Star 4
Fork 7

[Code](#)
[Issues 14](#)
[Pull requests 1](#)
[Actions](#)
[Projects](#)
[Wiki](#)
[Security](#)
[Insights](#)

[master](#)
[HELICS-Examples / user_guide_examples / advanced / advanced_brokers / simultaneous /](#)
Go to file
Add file
...

trevorhardy

Remove output graphs of simultaneous example

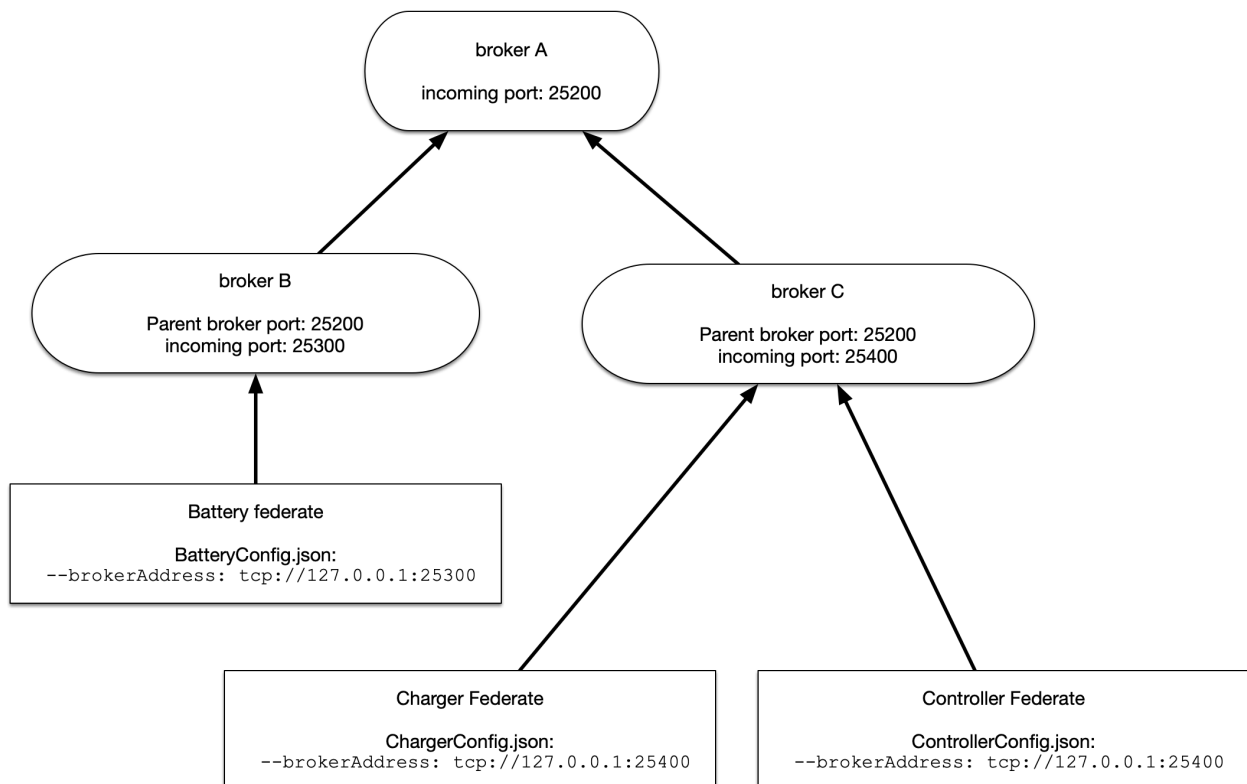
4bf3e89
on Nov 24, 2020
History

..		
folder federation_1	Remove output graphs of simultaneous example	2 months ago
folder federation_2	Remove output graphs of simultaneous example	2 months ago
folder federation_3	Remove output graphs of simultaneous example	2 months ago
file simultaneous_cosimulation_example.md	Add simultaneous co-simulation example	2 months ago

What is this co-simulation doing?

This example shows you how to configure a co-simulation to take advantage of multiple brokers. Though we'll be running this example on a single computer, the application of broker hierarchies is more common when running a co-simulation across multiple computers.

Differences compared to Advanced Default example



As will be shown, the use of multiple brokers will not affect the results of the co-simulation.

HELICS differences

Broker hierarchies are primarily used to help improve the performance of the co-simulation by allowing federates that interact strongly with each other to run on a single compute node, thereby allowing them to exchange information with each other quickly rather than over a relatively slow network connection to the rest of the federates on other compute nodes. This can be particularly helpful when the other compute nodes reside at off-site locations and the co-simulation communication is taking place between them over the public internet. (See the [User Guide section on broker hierarchies](#) for further details.)

Not all federations lend themselves to segregation like this; the example here doesn't really support such segregation as both the Battery and the Controller talk frequently with the Charger.

HELICS components

When implementing across compute nodes, the configuration is simpler will be simpler than in this example because the need to segregate the federates and brokers is only a function of IP address where HELICS can use the default port number on each compute node. To get this example to run on a single computer, the hierarchy must be implemented through the use of specific port numbers for specific brokers.

When running across multiple compute nodes, the relevant portion of the `helics_cli` runner files would look like this:

`broker_hierarchy_runner_A.json`

```
...
"exec": "helics_broker --loglevel=7 --timeout='10s' ",
...
```

`broker_hierarchy_runner_B.json` and `broker_hierarchy_runner_C.json`

```
...
"exec": "helics_broker -f <number of federates> --loglevel=7 --timeout='10s' --broker_
↪address=tcp://<IP address of broker A>",
...
```

Additionally, the federates would not need `brokerAddress` in their configuration since they would look on their local node for a broker by default.

Since this example *was* made to run on a single computer, we use the port number to segregate the federates and broker. `--port=` is use to define the port number on which the broker looks for connections to federates and `--broker_address=` is used to define the IP address with port of the parent broker. (The loopback address of 127.0.0.1 is used to look for the broker on the same node.)

Additionally, each federate has to define the broker to which it is attempting to connect by including the `brokerAddress` parameter in its own configuration; this allows for the definition of the port number the federate should use to connect to it's broker:

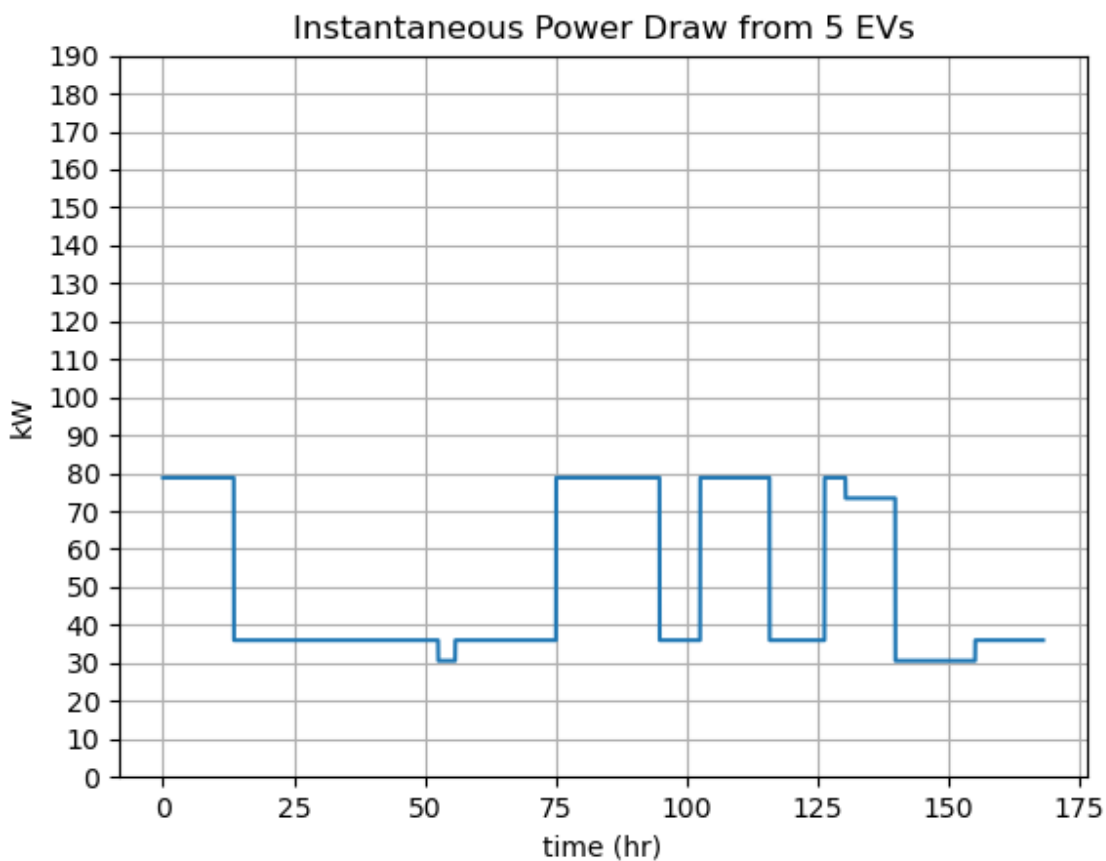
```
"brokerAddress": "tcp://127.0.0.1:25300",
```

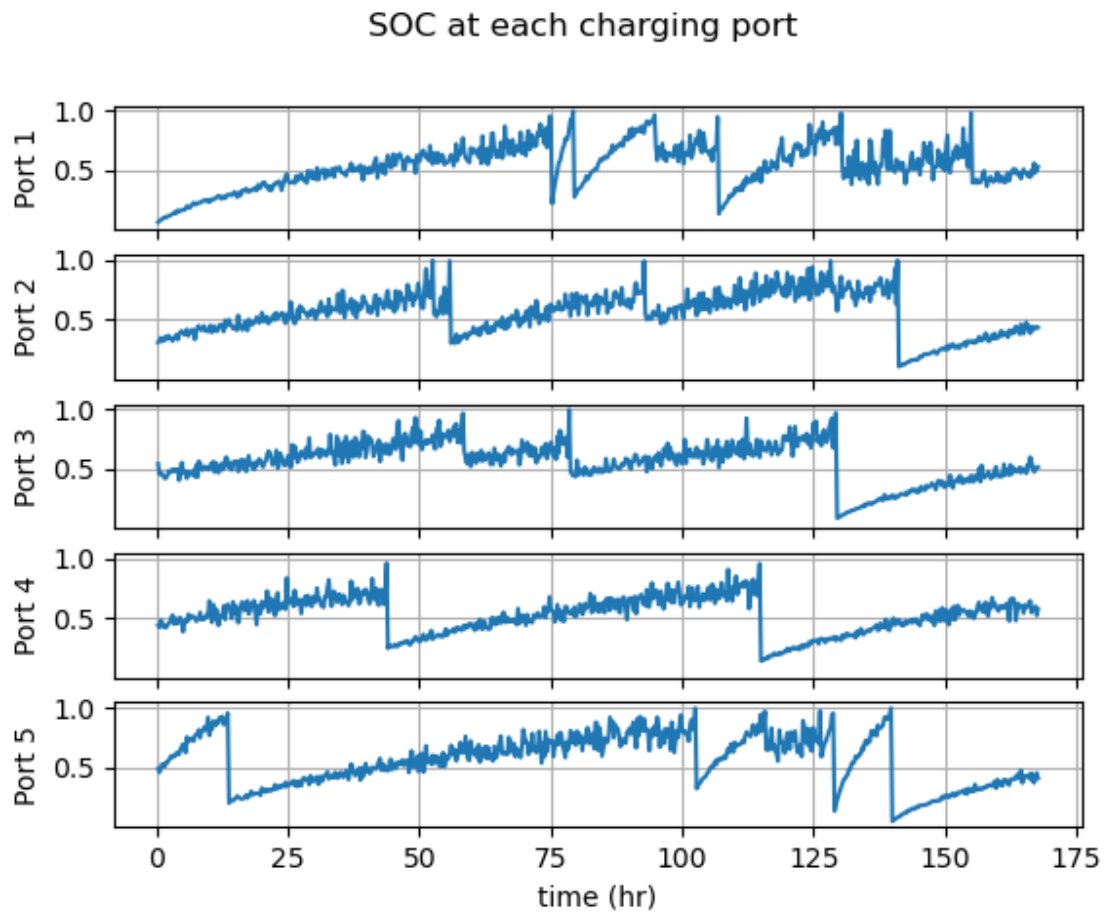
Execution and Results

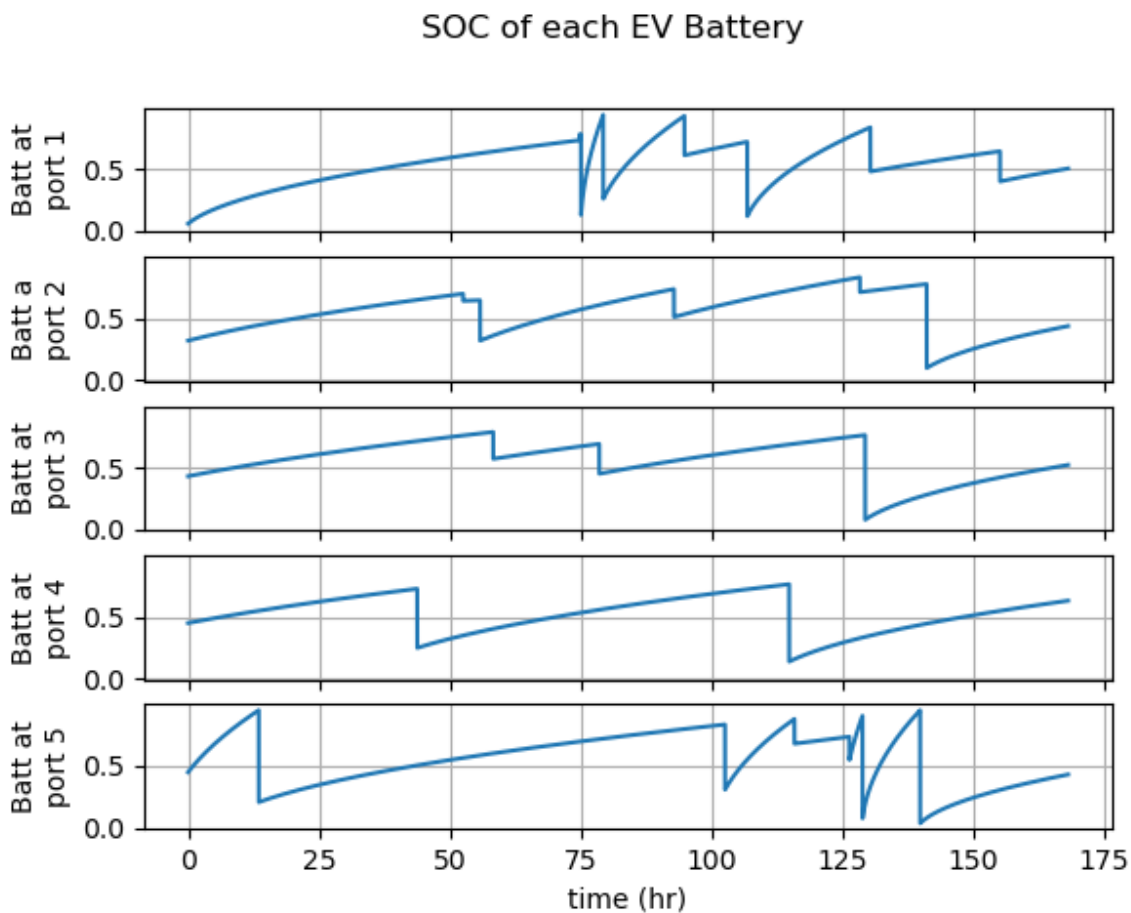
Since this example requires three brokers and their respective federates to run simultaneously, `helics_cli` will be used to launch the three sets of brokers and federates, just like the in [simultaneous co-simulation example](#)

```
$ helics run --path=./broker_hierarchy_runner_A.json & $ helics run --path=./broker_hierarchy_runner_B.json & $ helics run --path=./broker_hierarchy_runner_C.json &
```

The peak charging results are shown below. As can be seen, the peak power amplitude and the total time at peak power are impacted by the random number generator seed.







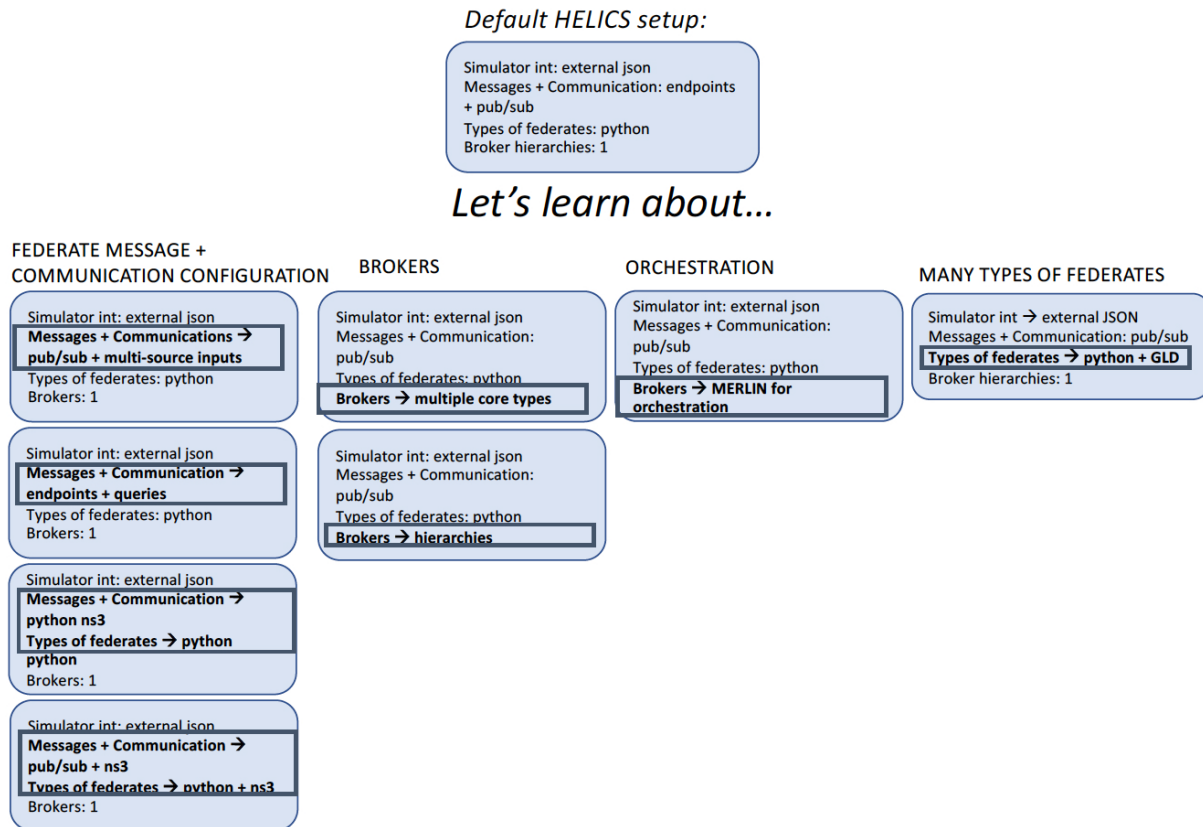
These results are identical to those in the base *Advanced Default example*; this is expected as only the structure of the co-simulation has been changed and not any of the federate code itself.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Brokers - Multi-Protocol Brokers







This example shows how to configure a HELICS co-simulation to implement a broker structure that utilizes multiple core types in a single co-simulation. Typically, all federates in a single federation use the same core type (ZMQ by default) but HELICS can be set up to utilize different core types in the same federation

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Fundamental Examples*
 - * *HELICS Differences*
 - *HELICS Components*
- *Execution and Results*

Where is the code?


This example on [multiple brokers](#) can be found [here](#). If you have issues navigating the examples, visit the [HELICS Gitter page](#) or the [user forum on GitHub](#).











[GMLC-TDC / HELICS-Examples](#)

 Watch 6
  Star 4
  Fork 7

[Code](#)
[Issues 14](#)
[Pull requests 1](#)
[Actions](#)
[Projects](#)
[Wiki](#)
[Security](#)
[Insights](#)

[master](#)
[HELICS-Examples / user_guide_examples / advanced / advanced_brokers / multi_broker /](#)
[Go to file](#)
[Add file](#)
[...](#)


 trevorhardy Add graphing of outputs to multibroker example
 7e0bb54 on Nov 24, 2020
[History](#)

...		
 Battery.py	Add graphing of outputs to multibroker example	2 months ago
 BatteryConfig.json	Update multibroker example for v3.0.0-alpha.2	2 months ago
 Charger.py	Add graphing of outputs to multibroker example	2 months ago
 ChargerConfig.json	Update multibroker example for v3.0.0-alpha.2	2 months ago
 Controller.py	Add graphing of outputs to multibroker example	2 months ago
 ControllerConfig.json	Update multibroker example for v3.0.0-alpha.2	2 months ago
 multi_broker_config.json	Update multibroker example for v3.0.0-alpha.2	2 months ago
 multi_broker_example.md	Update multibroker example for v3.0.0-alpha.2	2 months ago
 multi_broker_runner.json	Update multibroker example for v3.0.0-alpha.2	2 months ago

What is this co-simulation doing?

This example shows you how to configure a co-simulation to use more than one core type in the same federation. The example itself has the same functionality as the Advanced Default example as the only change is a structural to the federation and not the federate code itself.

Differences compared to the Advanced Default example

For this example, the *Advanced Default example* has been split up so that each federate uses a different core type in a single federation.

HELICS differences

Typically, all federates in a federation use the same core type. There can be cases, though, where a multi-site co-simulation with a more complex networking environment or performance requirements dictate the need for some federates to utilize a difference core type than others. For example, the IPC core utilizes a Boost library function to allow two executables both using Boost to communicate between themselves when running on the same compute node; since this is in-memory communication rather than over the network stack, performance is expected to be higher. It could be that a particular federation has been optimized to take advantage of this but must also communicate with federates on a separate compute node via ZMQ. In this case, a so-called “multibroker” can be configured to allow for the federation to run. (See the User Guide section on the *multi-protocol broker* and *broker core types* for further details.)

In this example, we won’t be doing anything like that but, for demonstration purposes, simply using the same federation from the *Advanced Default example*. and configuring it so each federate uses a different core type.

HELICS Components

To configure a multibroker, the broker configuration line is slightly extended from a traditional federation. From the `helics_cli` runner configuration file `multi_broker_runner.json`

```
...
"exec": "helics_broker -f 3 --coreType=multi --config=multi_broker_config.json --
↪name=root_broker",
...
```

The `coreType` of the broker is set to `multi` and a configuration file is specified. That file looks like this:

```
{
  "master": {
    "coreType": "test"
  },
  "comms": [
    {
      "coreType": "zmq",
      "port": 23500
    },
    {
      "coreType": "tcp",
      "port": 23700
    },
    {
      "coreType": "udp",
      "port": 23900
    }
  ]
}
```

The first and most important note: `master` and `comms` are reserved words in this context and **MUST** be used. The `master` core type must be `test` but the core types for the federates can be any of the supported cores. Again, as in *other similar* examples, because we are running this on a single compute node, the port for each core type must be specified and the federates using those core types need to have the `brokerPort` property set to the corresponding core's port number.

BatteryConfig.json

```
...
"name": "Battery",
"loglevel": 1,
"coreType": "zmq",
"brokerPort": 23500,
...
```

ChargerConfig.json

```
...
"name": "Charger",
"loglevel": 1,
"coreType": "tcp",
"brokerPort": 23700,
...
```

ControllerConfig.json

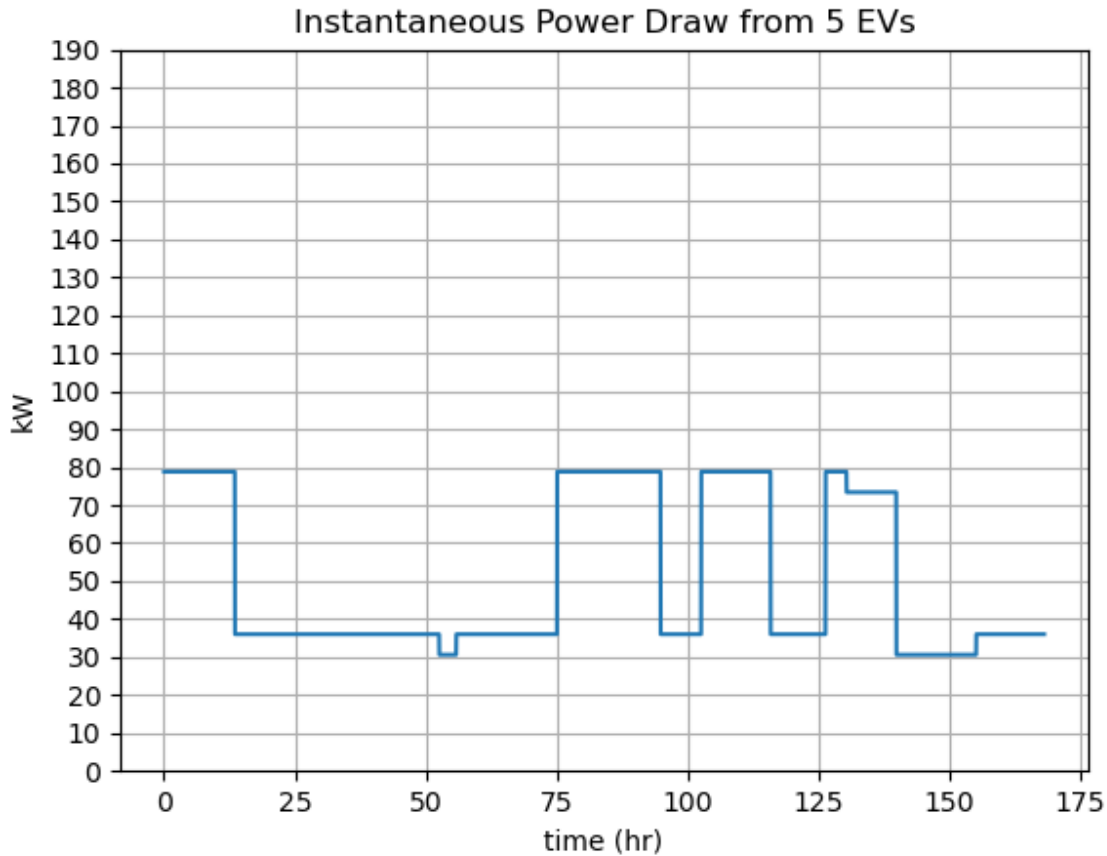
```
...  
"name": "Controller",  
"loglevel": 1,  
"coreType": "udp",  
"brokerPort": 23900,  
...
```

Execution and Results

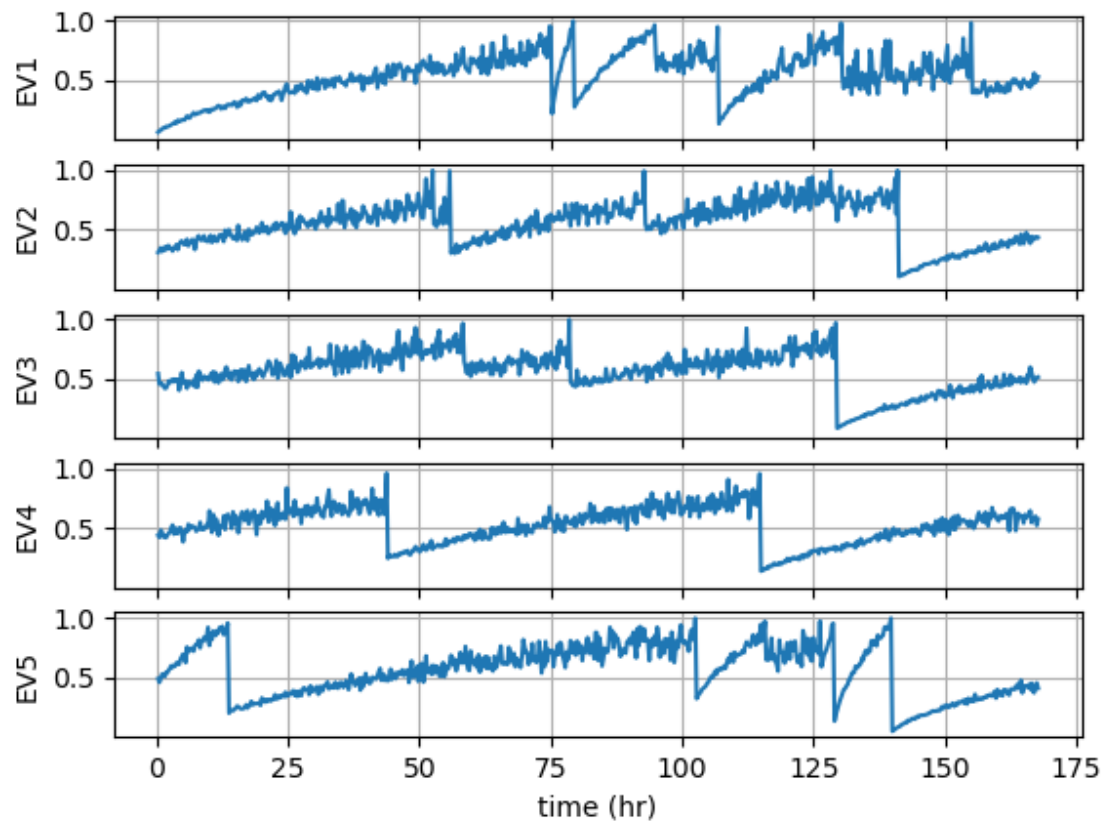
Unlike the other advanced broker examples, this one can be run with a single `helics_cli` command:

```
$ helics run --path=./multi_broker_runner.json
```

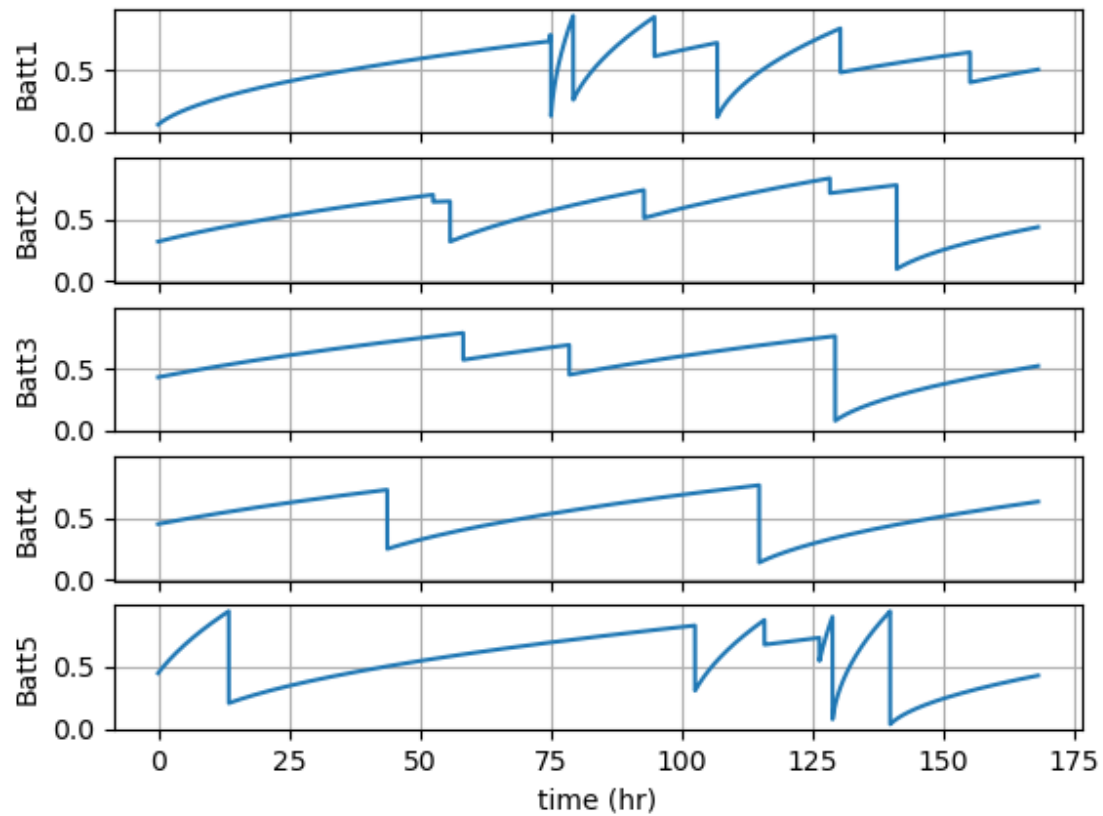
As has been mentioned, since this is just a change to the co-simulation architecture, the results are identical to those in the [Advanced Default example](#).



SOC at each charging port



SOC of each EV Battery

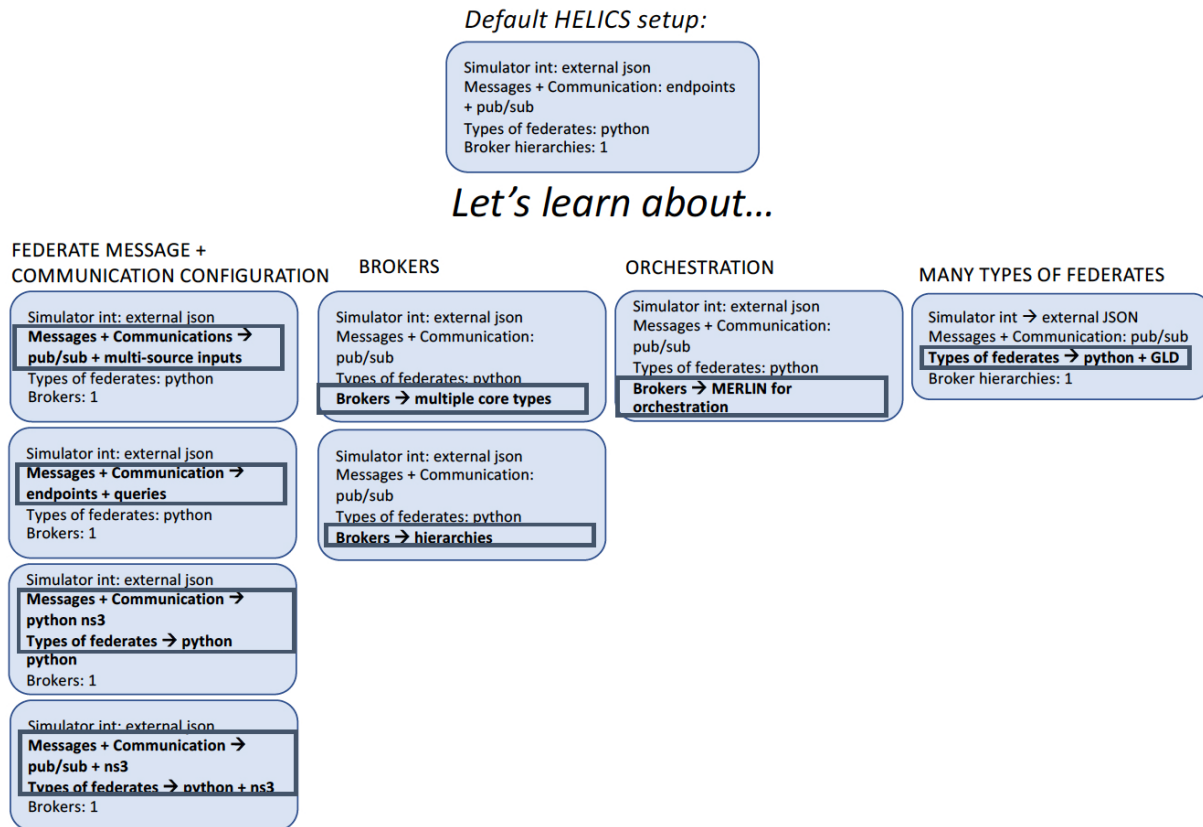


Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Federation Queries







This demonstrates the use of federation queries and performs dynamic configuration by using the information from the query to configure the Battery federate.

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Fundamental Examples*
 - * *HELICS Differences*
 - *HELICS Components*
- *Execution and Results*

Where is the code?


This example on queries can be found [here](#). If you have issues navigating the examples, visit the [HELICS Gitter page](#) or the [user forum on GitHub](#).


GMLC-TDC / HELICS-Examples

 Watch
  Star
  Fork

<> Code
 14 Issues
 1 Pull requests
 Actions
 Projects
 Wiki
 Security
 Insights

master
 HELICS-Examples / user_guide_examples / advanced / advanced_message_comm / query /
 Go to file
 Add file
 ...


trevorhardy Reduce logging level and add a few logging messages to clarify the log
 0a74192 on Nov 24, 2020 History

..		
Battery.py	Reduce logging level and add a few logging messages to clarify the log	2 months ago
BatteryConfig.json	Reduce logging level and add a few logging messages to clarify the log	2 months ago
Charger.py	Update query, output graph file names, and helics runner JSON name	2 months ago
ChargerConfig.json	Reduce logging level and add a few logging messages to clarify the log	2 months ago
Controller.py	Update query, output graph file names, and helics runner JSON name	2 months ago
ControllerConfig.json	Reduce logging level and add a few logging messages to clarify the log	2 months ago
query_example.md	Reduce logging level and add a few logging messages to clarify the log	2 months ago
query_runner.json	Reduce logging level and add a few logging messages to clarify the log	2 months ago

What is this co-simulation doing?

This example shows how to run queries on a federation and to use the output of the queries to configure a federate. Rather than a static configuration that is defined prior to runtime, this dynamic configuration can be useful for federations that change composition frequently.

Difference compared to the Advanced Default example

This example has the same federates interacting in the same ways as in the *Advanced Default example*. The only difference is the use of queries for dynamic configuration rather than static configuration.

HELICS Differences

In most of the examples presented here, the configuration of the federation is defined prior to executing the co-simulation via the configuration JSON files. It is possible, though, with extra effort and careful design, to write the federate code such that they self-configure based on the other participants in the co-simulation. This example provides a simple demonstration of this by having the Battery federate query the federation and look for the values that the Charger federate is publishing and subscribing to them.

HELICS components

Battery.py contains all the changes from the *Advanced Default example* that allow it to perform dynamic configuration. Specifically:

- Sleeping a few seconds to ensure all other federates in the federation have configured so that the Battery federate can query the federation and know it is seeing the comprehensive configuration.

```
sleep_time = 5
logger.debug(f"Sleeping for {sleep_time} seconds")
time.sleep(sleep_time)
```

- `eval_data_flow_graph` is a new function that performs a data graph query on the federation. This query evaluates the connections between federates, showing who is publishing and subscribing to what. The function also takes the output from the query and parses it into a form that can be easily used for the dynamic configuration.

```
def eval_data_flow_graph(fed):
    query = h.helicsCreateQuery("broker", "data_flow_graph")
    graph = h.helicsQueryExecute(query, fed)
```

- The Battery federate subscribes to all the publications from the Charger federate based on the results of the data flow graph.

```
for core in graph["cores"]:
    if core["federates"][0]["name"] == "Charger":
        for pub in core["federates"][0]["publications"]:
            key = pub["key"]
            sub = h.helicsFederateRegisterSubscription(fed, key)
            logger.debug(f"Added subscription {key}")
```

- After making the subscription, the Battery federate re-evaluates the data flow graph and updates its own internal record of the configuration

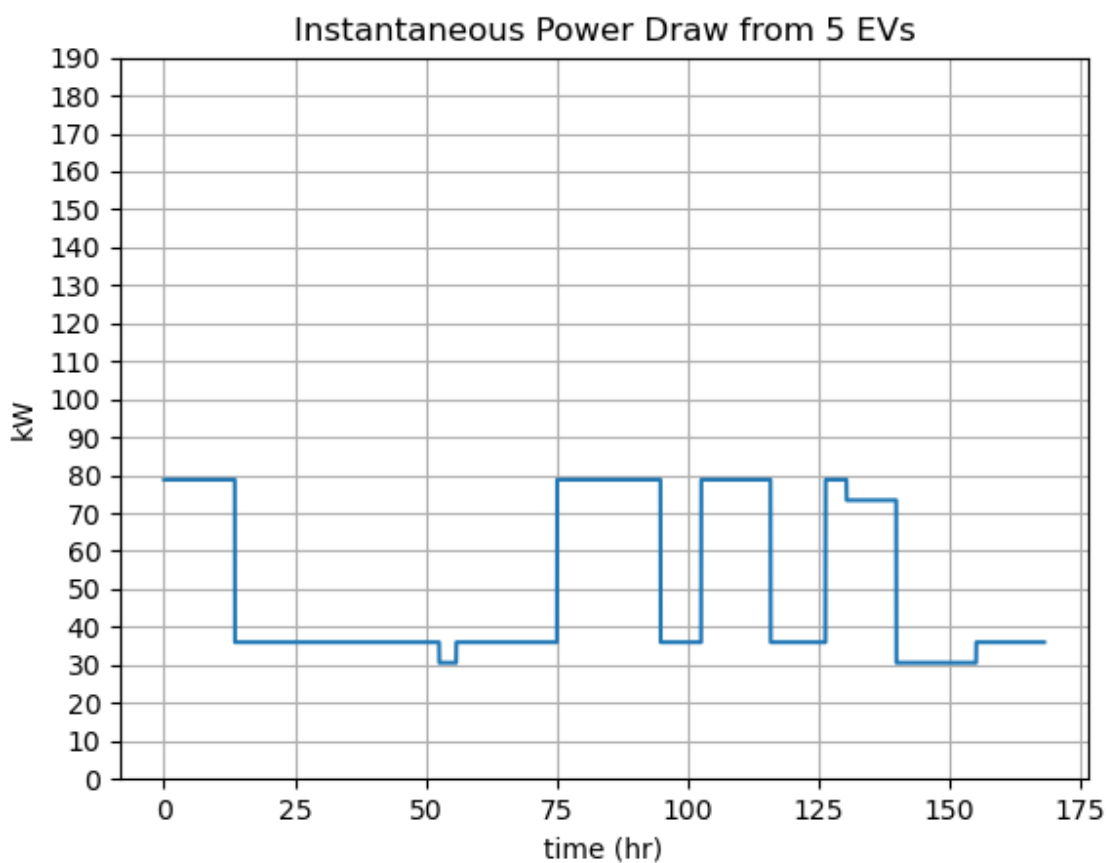
```
# The data flow graph can be a time-intensive query for large
# federations
# Verifying dynamic configuration worked.
graph, federates_lut, handle_lut = eval_data_flow_graph(fed)
# logger.debug(pp.pformat(graph))
sub_count = h.helicsFederateGetInputCount(fed)
logger.debug(f"Number of subscriptions: {sub_count}")
subid = {}
sub_name = {}
for i in range(0, sub_count):
    subid[i] = h.helicsFederateGetInputByIndex(fed, i)
    sub_name[i] = h.helicsSubscriptionGetKey(subid[i])
    logger.debug(f"\tRegistered subscription---> {sub_name[i]}")
```


Execution and Results

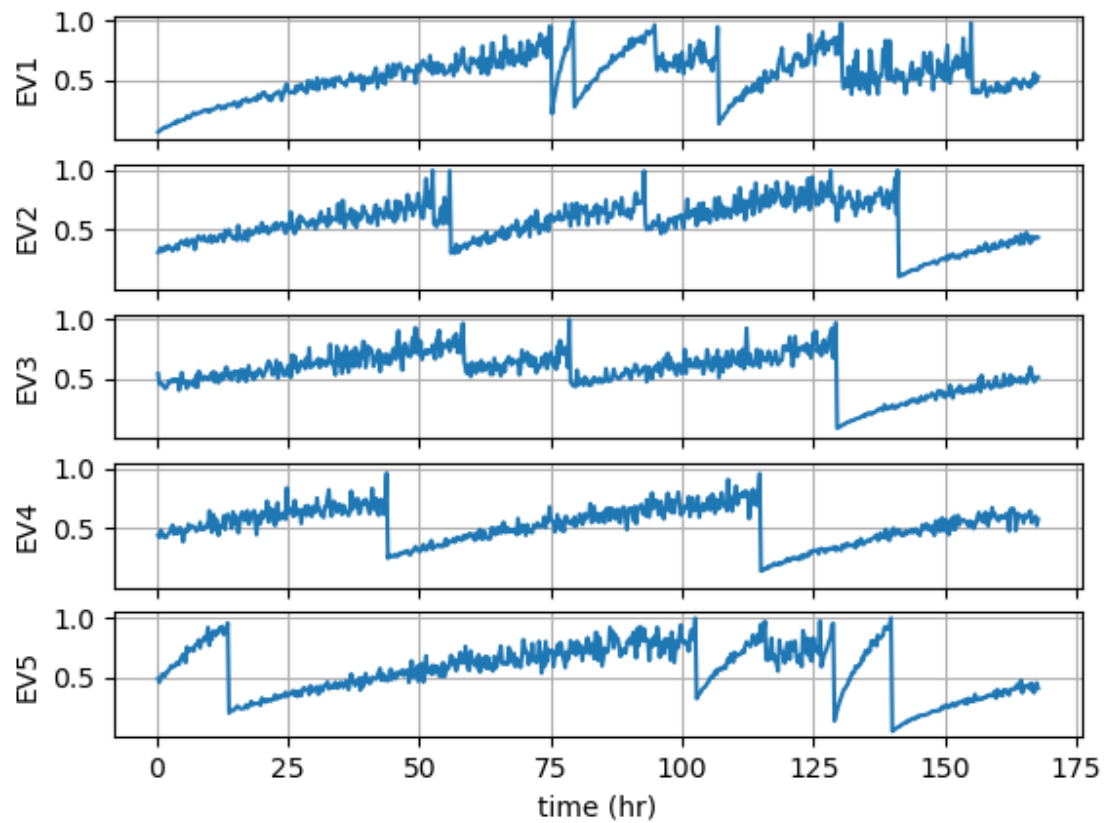
Run the co-simulation:

```
$ helics run --path=./multi_broker_runner.json
```

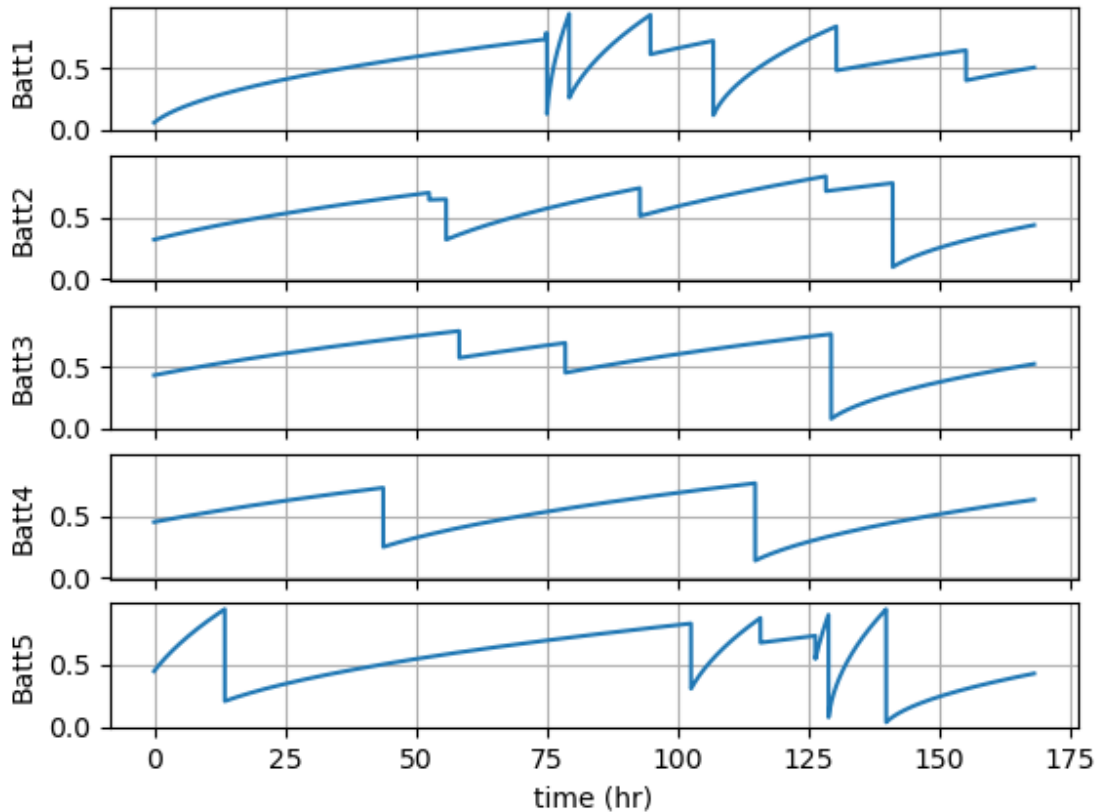
Since this is only a change to the configuration method of the federation, the results are identical to those in the [Advanced Default example](#).



SOC at each charging port



SOC of each EV Battery



The dynamic configuration can also be seen by looking at the log file for the Battery federate (`Battery.log`). The pre-configure data flow graph only showed five subscriptions, all made by the Charger federate of the Battery federates current

```
Pre-configure data-flow graph query.
Federate Charger (with id 131072) has the following subscriptions:
  Battery/EV1_current from federate Battery
  Battery/EV2_current from federate Battery
  Battery/EV3_current from federate Battery
  Battery/EV4_current from federate Battery
  Battery/EV5_current from federate Battery
Added subscription Charger/EV1_voltage
```

After analyzing the results of the data-flow graph, the Battery federate subscribing to the appropriate publications from the Charger federate, and re-running the query, the subscription list looks like this:

```
Post-configure data-flow graph query.
Federate Battery (with id 131073) has the following subscriptions:
  Charger/EV1_voltage from federate Charger
  Charger/EV2_voltage from federate Charger
  Charger/EV3_voltage from federate Charger
  Charger/EV4_voltage from federate Charger
  Charger/EV5_voltage from federate Charger
Federate Charger (with id 131072) has the following subscriptions:
```

(continues on next page)

(continued from previous page)

```
Battery/EV1_current from federate Battery
Battery/EV2_current from federate Battery
Battery/EV3_current from federate Battery
Battery/EV4_current from federate Battery
Battery/EV5_current from federate Battery
```

Questions and Help

Do you have questions about HELICS or need help?

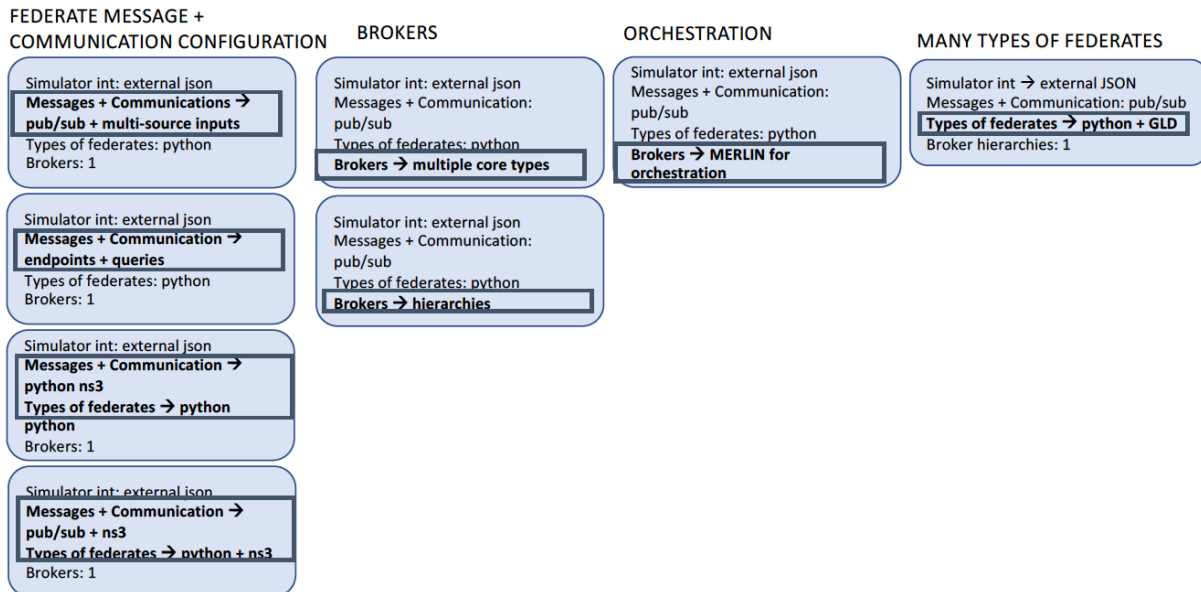
1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Multi-Input

Default HELICS setup:

```
Simulator int: external json
Messages + Communication: endpoints
+ pub/sub
Types of federates: python
Broker hierarchies: 1
```

Let's learn about...



This demonstrates the use of federation queries and performs dynamic configuration by using the information from the query to configure the Battery federate.

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Fundamental Examples*

- * *HELICS Differences*
- *HELICS Components*
- *Execution and Results*

Where is the code?

This example on multi-inputs can be found [here](#). If you have issues navigating the examples, visit the [HELICS Gitter](#) page or the user forum on [GitHub](#).

The screenshot shows the GitHub repository **GMLC-TDC / HELICS-Examples**. The repository has 6 watches, 4 stars, and 7 forks. The navigation bar includes links for Code, Issues (14), Pull requests (1), Actions, Projects, Wiki, Security, and Insights. The current view is the **master** branch, showing the file structure: **HELICS-Examples / user_guide_examples / advanced / advanced_message_comm / multi_input /**. Below the file structure, a commit history table is displayed for the commit **bc63745** on Nov 25, 2020, by **trevorhardy**. The commit message is "Add file save lines to Charger for multi-input example".

File	Commit Message	Time Ago
Battery.py	Update documentation for the get_new_battery function	2 months ago
BatteryConfig.json	new file structure for user guide examples	3 months ago
Charger.py	Add file save lines to Charger for multi-input example	2 months ago
ChargerConfig.json	new file structure for user guide examples	3 months ago
multi_input_example.md	Update for v3.0.0-alpha.2	2 months ago
multi_input_runner.json	Update for v3.0.0-alpha.2	2 months ago

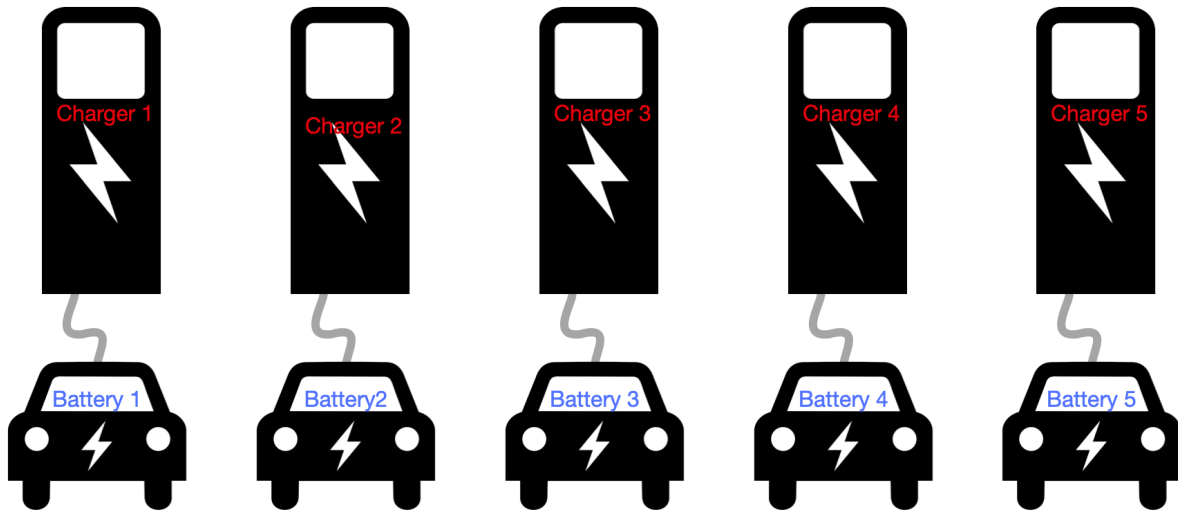
What is this co-simulation doing?

This example shows how to use inputs, allowing multiple publications to arrive at the same input handle (similar to a subscription, as you'll see) and a demonstration on one method of managing data conflicts that can arise.

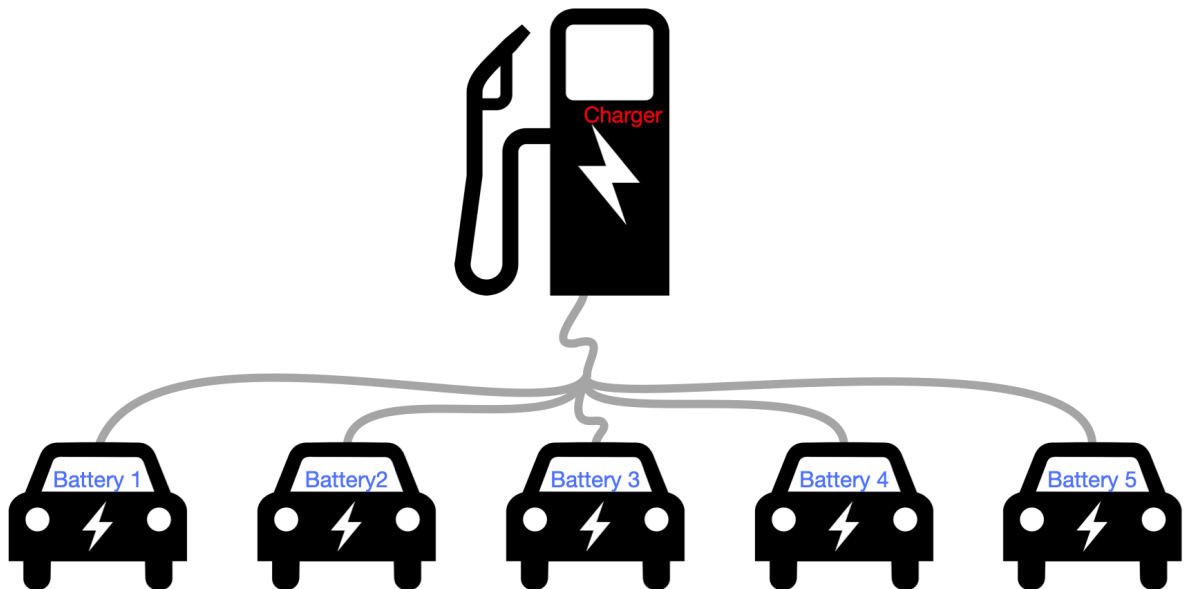
Difference compared to the Advanced Default example

This example deviates fairly significantly from the *Advanced Default example* in that it only has a Battery and Charger federate. The Charger federate was modeled with one charging terminal that branches out to the five Battery terminals. That is, from the Charger federates perspective, there is only one charging voltage and one charging current even though the federation is still constructed to charge five batteries.

Advanced Default



Multi-Input



The difference in the model is entirely implied by the HELICS configuration; the physics of the system is modeled through the configuration and this is one valid interpretation.

Additionally, since the protocol (to use the term loosely) in the Advanced Default example for a Battery indicating it was fully charged and the Charger confirming was the removal of the charging voltage, this simple protocol won't work as written when using multi-inputs. Rather than implementing a more sophisticated protocol and to keep the code simple, we decided to just charge one battery at each terminal over the duration of the simulation. Also, due to the removal of the protocol, there is no need of the Controller federate to determine when to stop charging the battery as

the batteries self-terminate their charging.

HELICS differences

With a single charger being used to charge five batteries, each battery still publishes it's charging current as before but only has one subscription, the single charging voltage. This shows up on the Battery federate configuration:

BatteryConfig.json

```
{
  "name": "Battery",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": true,
  "publications": [
    {
      "key": "Battery/EV1_current",
      "type": "double",
      "unit": "A",
      "global": true
    },
    {
      "key": "Battery/EV2_current",
      "type": "double",
      "unit": "A",
      "global": true
    },
    ...
  ],
  "subscriptions": [
    {
      "key": "Charger/EV_voltage",
      "type": "double",
      "unit": "V",
      "global": true
    }
  ]
}
```

The Charger federate configuration is also altered, using an `input` rather than a `subscription` handle to allow all publications from the Battery federates to be received on one handle. The input has been configured to allow multiple inputs and lists the publications that should be targeted toward it and to handle these multiple inputs by summing them.

```
{
  "name": "Charger",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
```

(continues on next page)

(continued from previous page)

```
"terminate_on_error": true,
"publications": [
  {
    "key": "Charger/EV_voltage",
    "type": "double",
    "unit": "V",
    "global": true
  }
],
"inputs": [
  {
    "key": "Battery/charging_current",
    "type": "double",
    "global": true,
    "multi_input_handling_method": "sum",
    "targets": [
      "Battery/EV1_current",
      "Battery/EV2_current",
      "Battery/EV3_current",
      "Battery/EV4_current",
      "Battery/EV5_current"
    ]
  }
]
}
```

HELICS Components

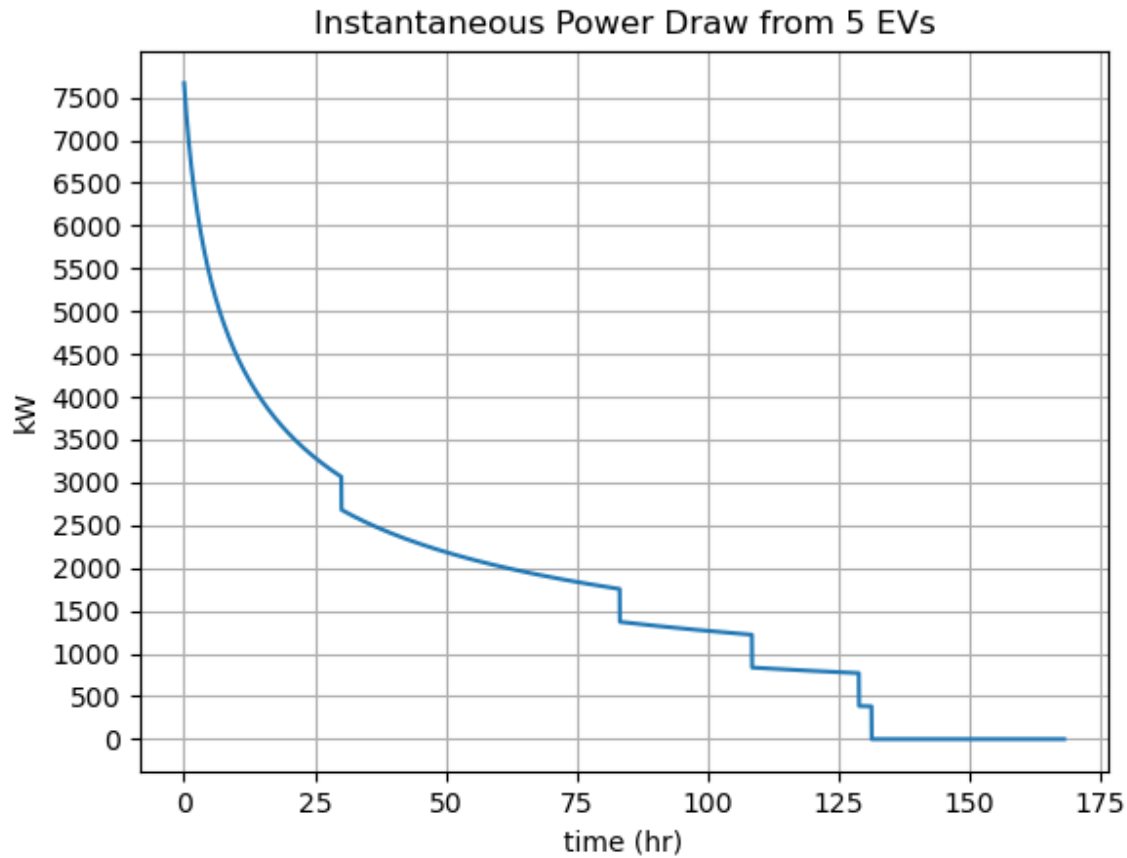
Battery.py and Charger.py have both been simplified such that only battery per charging terminal is charged over the duration of the simulation. When the battery reaches full SOC, it self-terminates charging.

Execution and Results

Run the co-simulation:

```
$ helics run --path=./multi_input_runner.json
```

The primary result of interest is still the cumulative charging power.



As the batteries are not replaced during charging, the initial charging power will be the peak power. The points in time when a battery reaches full charge, though, can be clearly seen as the discrete changes in cumulative charging power.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Monte Carlo Co-Simulations

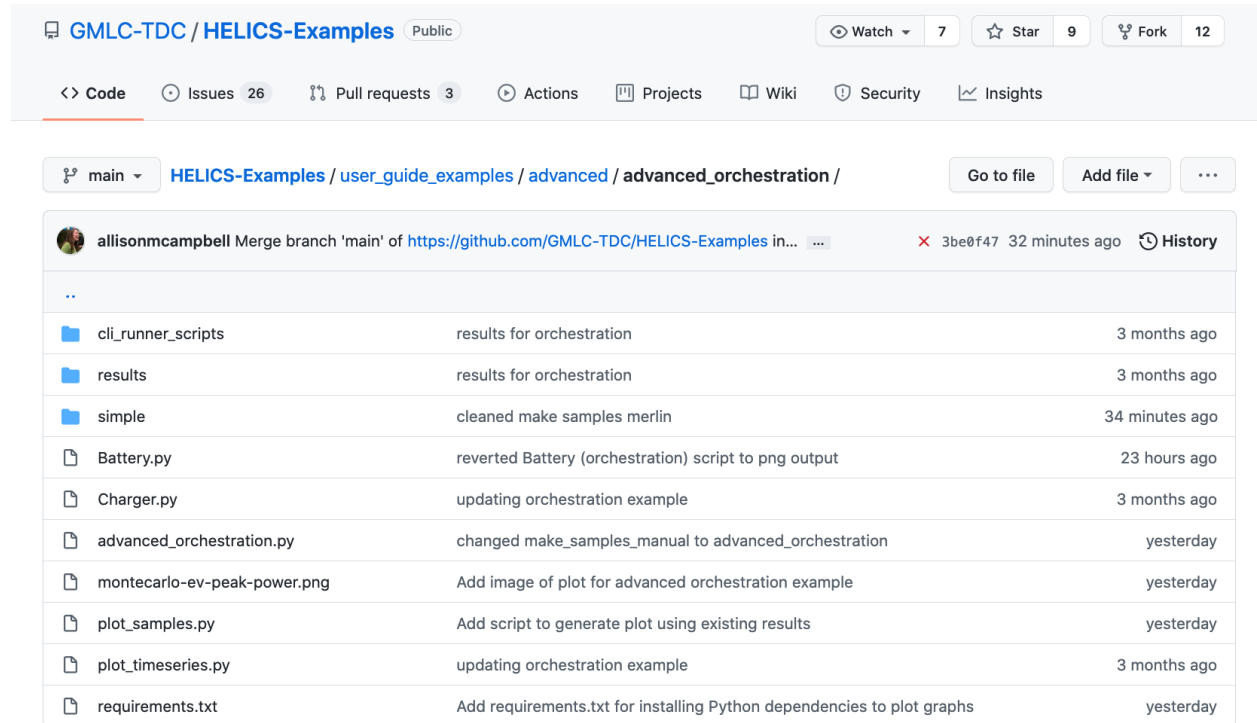
This tutorial will walk through how to set up a HELICS Monte Carlo simulation using two techniques: (1) in series on a single machine, and (2) in parallel on an HPC cluster using Merlin. We assume that you have already completed the [orchestration tutorial with Merlin](#) and have some familiarity with how Merlin works.

- *Where is the code?*
- *What is this Co-simulation doing?*
- *Probabilistic Uncertainty Estimation*
- *Execution and Results*

- *Manual Orchestration Execution*
- *Merlin Orchestration Execution*

Where is the code?

Code for the Monte Carlo simulation and the full Merlin spec can be found in the [HELICS Examples Repo](#). If you have issues navigating the examples, visit the [HELICS Gitter](#) page or the [user forum on GitHub](#).



Public

Watch 7 Star 9 Fork 12

Code Issues 26 Pull requests 3 Actions Projects Wiki Security Insights

main HELICS-Examples / user_guide_examples / advanced / advanced_orchestration / Go to file Add file ...

allisonmcampbell Merge branch 'main' of https://github.com/GMLC-TDC/HELICS-Examples in... 3be0f47 32 minutes ago History

File	Commit Message	Time
cli_runner_scripts	results for orchestration	3 months ago
results	results for orchestration	3 months ago
simple	cleaned make samples merlin	34 minutes ago
Battery.py	reverted Battery (orchestration) script to png output	23 hours ago
Charger.py	updating orchestration example	3 months ago
advanced_orchestration.py	changed make_samples_manual to advanced_orchestration	yesterday
montecarlo-ev-peak-power.png	Add image of plot for advanced orchestration example	yesterday
plot_samples.py	Add script to generate plot using existing results	yesterday
plot_timeseries.py	updating orchestration example	3 months ago
requirements.txt	Add requirements.txt for installing Python dependencies to plot graphs	yesterday

The necessary files are:

- Python program for Battery federate ([Battery.py](#))
- Python program for Charger federate ([Charger.py](#))
- Python program to generate helics_cli JSON files and execute ([advanced_orchestration.py](#))

What is this co-simulation doing?

This example walks through how to set up a probabilistic model with Monte Carlo simulations. This Monte Carlo co-simulation is built from a simple two federate example, based on the [Endpoint Federates Example](#). In this example, there is a Charger federate which publishes voltage and a Battery federate which publishes current.

All of the HELICS configurations are the same as in the Endpoint example. The internal logic of the federates has been changed for this implementation. The Charger federate assumes the role of *deciding* if the Battery should continue to charge. The Battery sends a message of its current SOC (state of charge), a number between 0 and 1. If the SOC is less than 0.9, the Battery is instructed to continue charging, otherwise, it is instructed to cease charging. The Battery federate has all the logic internal for adding energy and selecting a new “battery” (charging rate) if the SOC is deemed sufficient. Energy is added to the “battery” according to the previous time interval and the charge rate of the battery. In this way, the only stochastic component to the system is the **selected charge rate**. For example, the Endpoint Example

allowed the Battery federate to randomly select batteries of different sizes, and the Charger to select charge rates from a list of options. In this implementation, the battery size (capacity in kWh) is constant.

This simplification allows us to isolate a single source of uncertainty: the charge rate.

The co-simulation relies on stochastic sampling of distributions – an initial selection of vehicles for the EV charging garage. We want to ensure that we are not overly reliant on any one iteration of the co-simulation. To manage this, we can run the co-simulation N times, or a Monte Carlo co-simulation. The result will be a *posterior distribution* of the instantaneous power draw over a desired period of time.

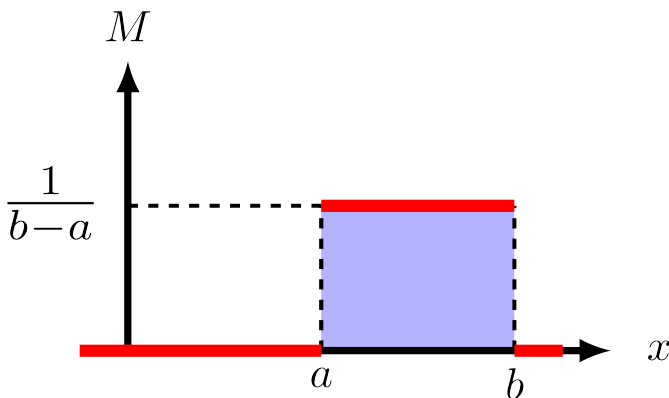
Probabilistic Uncertainty Estimation

A Monte Carlo simulation allows a researcher to sample random numbers repeatedly from a predefined distribution to explore and quantify uncertainty in their analysis. Additional detail about Monte Carlo methods can be found on [Wikipedia](#) and [MIT Open Courses](#).

In a Monte Carlo co-simulation, a probability distribution of possible values can be used in the place of **any** static value in **any** of the simulators. For example, a co-simulation may include a simulator (federate) which measures the voltage across a distribution transformer. We can quantify measurement error by replacing the deterministic (static) value of the measurement with a random value from a uniform distribution. Probabilistic distributions are typically described with the following notation:

$$M \sim U(a, b)$$

Where M is the measured voltage, a is the lower bound for possible values, and b is the upper bound for possible values. This is read as, “ M is distributed uniformly with bounds a and b .”



The uniform distribution is among the most simple of probability distributions. Additional resources on probability and statistics are plentiful; [Statistical Rethinking](#) is highly recommended.

Monte Carlo Co-sim Example: EV Garage Charging

The example co-simulation to demonstrate Monte Carlo distribution sampling is that of an electric vehicle (EV) charging garage. Imagine a parking garage that only serves EVs, has a static number of charging ports, and always has an EV connected to a charging port. An electrical engineer planning to upgrade the distribution transformer prior to building the garage may ask the question: What is the likely power draw that EVs will demand?

Probability Distributions

Likely is synonymous for *probability*. As we are interested in a probability, we cannot rely on a deterministic framework for modeling the power draw from EVs. I.e., we cannot assume that we know a priori the exact demand for Level 1, Level 2, and Level 3 chargers in the garage. A deterministic assumption would be equivalent to stating, e.g., that 30% of customers will need Level 1 charge ports, 50% will need Level 2, and 20% will need Level 3. What if, instead of static proportions, we assign a distribution to the need for each level of charge port. The number of each level port is discrete (there can't be 0.23 charge ports), and we want the number to be positive (no negative charge ports), so we will use the [Poisson distribution](#). The Poisson distribution is a function of the anticipated average of the value and the number of samples k . Then we can write the distribution for the number of chargers in each level, L , as

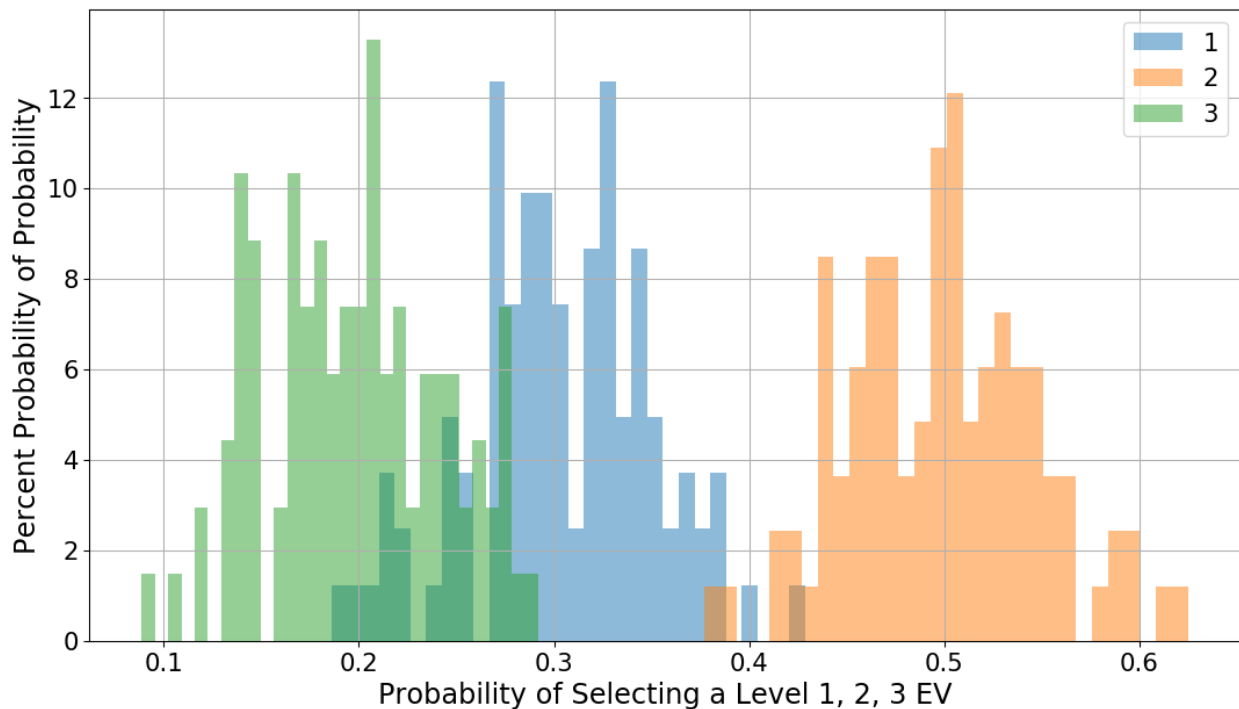
$$L \sim P(k,)$$

Let's extend our original assumption that the distribution of chargers is static to Poisson distributed, and let's assume that there are 100 total charging ports:

$$L1 \sim P(100,0.3)$$

$$L2 \sim P(100,0.5)$$

$$L3 \sim P(100,0.2)$$



What if we weren't entirely certain that the average values for $L1$, $L2$, $L3$ are 0.3, 0.5, 0.2, we can also sample the averages from a normal distribution centered on these values with reasonable standard deviations. We can say that:

$$\sim N(,)$$

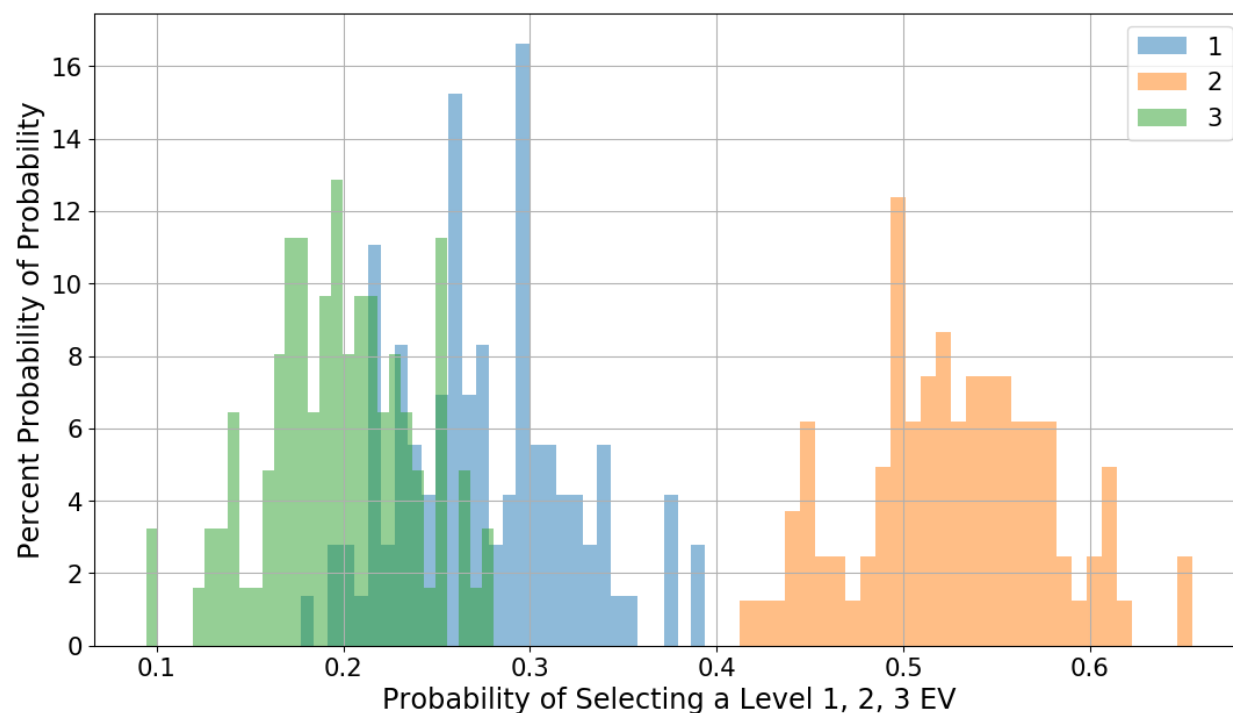
Which means that the input to L is distributed normally with average and standard deviation .

Our final distribution for modeling the anticipated need for each level of charging port in our $k = 100$ EV garage can be written as:

$$L \sim P(k,)$$

$$\sim N(,)$$

$L1$	$L2$	$L3$
0.3	0.5	0.2
$\sim N(1,3)$	$\sim N(1,2)$	$\sim N(0.05,0.25)$



Notice that the individual overplotted distributions in the two histograms above are different – there is more flexibility encoded into the second. The distributions in the second plot describe the following assumptions about the anticipated need for Level 1, 2, and 3 chargers:

1. The number of charging ports is discrete and positive (Poisson).
2. The variance in the number of ports should be normally distributed.
3. The average percentage of Level 1, 2, and 3 chargers is 30%, 50%, and 20%, respectively.
4. The variance around the average is uniformly distributed.
5. The variance is relatively very broad for Level 1, broad for Level 2, and very constrained for Level 3. This means we are very confident in our guess of the average percentage for Level 3 chargers, less confident for Level 2, and much less confident for Level 3.

We have described the individual distributions for each level of charging port. What we don't know, prior to running the Monte Carlo co-simulation, is how these distributions will jointly impact the research question.

Research Question Configuration

We want to address the research question: What is the likely power draw that EVs will demand?

At the beginning of the co-simulation, the distributions defined above will be sampled N times within the EV federate, where N = the number of parking spots/charging ports in the garage. The output of the initial sampling is the number of requested Level 1, 2, and 3 charging ports. The SOC for the batteries on board are initialized to a uniform random number between 0.05 and 0.5, and these SOC are sent to the Charger federate. If the SOC of an EV battery is less than 0.9, the EV Controller federate tells the EV battery in the EV federate to continue charging. Otherwise, the EV Charger disconnects the EV battery, and instructs the it to sample a new EV battery from the distributions (one sample).

After the two federates pass information between each other – EV Battery sends SOC, EV Charger instructs whether to keep charging or resample the distributions – the EV Battery calculates the total power demanded in the last time interval.

Execution and Results

Execution can be done with either a simple script (provided on the repo), or with Merlin.

Manual Orchestration Execution

Manual implementation of the co-simulation is done with the helper script `advanced_orchestration.py`, with command line execution:

```
$ python advanced_orchestration.py
```

This implementation will run a default co-simulation. The default parameters are:

```
samples = 30
output_path = os.getcwd()
numEVs = 10
hours = 24
plot = 0
run = 1
```

This means that we are generating 30 JSON files with unique seeds, we are using the current operating directory as the head for the output path, we are simulating 10 EVs in the co-simulation for one day, we are not running individual plots for each simulation, and we are executing the JSON files with `helics_cli` after they have been created.

If we wanted to run a Monte Carlo co-sim with different parameters, this would be:

```
$ python advanced_orchestration.py 10 . 100 24*7 0 0
```

This execution would create 10 JSON files with unique seeds, set the current directory as the head for the output path, simulate 100 EVs for a week, not generate plots with each simulation, and not execute the JSON files with `helics_cli` (meaning that `helics_cli` will not be called automatically – the user will need to manually execute `helics_cli` for each JSON file).

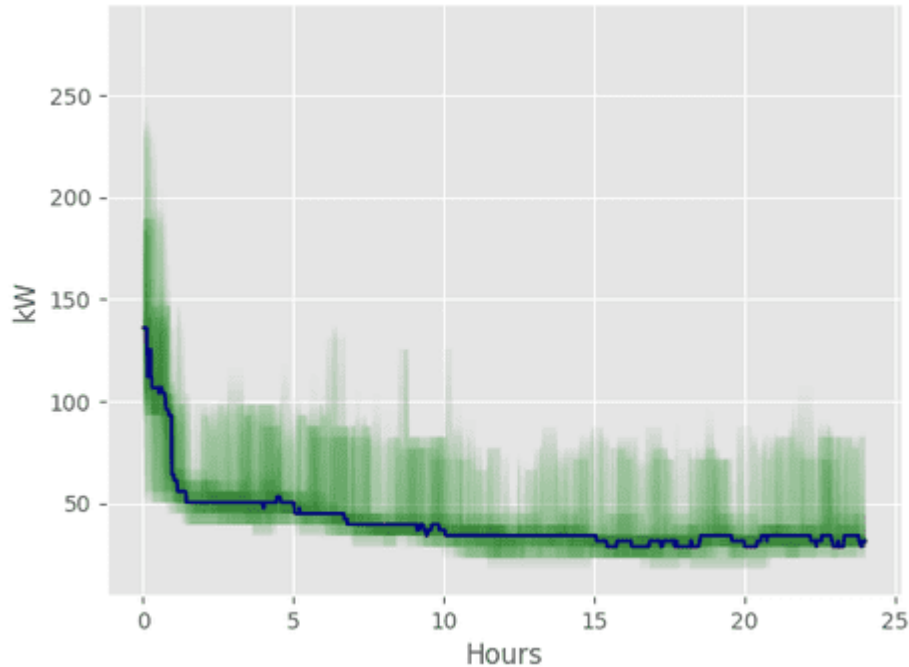
You may decide to adapt `advanced_orchestration.py` to suite your needs within the Merlin environment, in which case you would only need the helper script to create the JSON files. If you elect to execute `helics_cli` for the generated runner JSON files using the helper script, subdirectories are created for the `helics_cli` runner JSON files and for the csv results. Results for the default simulation are in the repo and can be used for confirming accurate execution.

```
out_json = output_path + "/cli_runner_scripts"
out_data = output_path + "/results"
```

In the runner scripts directory, there will be 30 JSON files. Each will have a unique seed parameter, otherwise they will all look identical:

```
{
  "federates": [
    {
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "helics_broker --federates=2 --loglevel=data --coretype=tcps --port 12345
↪",
      "host": "localhost",
      "loglevel": "data",
      "name": "broker_0"
    },
    {
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "python3 Battery.py --port 12345 --seed 10 --numEVs 10 --hours 24 --plot 0_
↪--outdir [path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]/results",
      "host": "localhost",
      "loglevel": "data",
      "name": "Battery_0"
    },
    {
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "python3 Charger.py --port 12345 --numEVs 10 --hours 24",
      "host": "localhost",
      "loglevel": "data",
      "name": "Charger_0"
    }
  ],
  "name": "Generated by advanced orchestration"
}
```

The final result of the default Monte Carlo co-simulation is shown below.



This is a time series density plot. Each simulation is a green line, and the blue solid line is the median of all simulations. From this plot, we can see that (after the system [initializes](#), after a few hours) the maximum demand from EVs in the garage will be roughly 125 kW. We could improve the analysis by conducting an initialization step and by running the simulation for a longer time period. This type of analysis provides the engineer with information about the probability that demand for power from N EVs will be X kW. The most commonly demanded power is less than 50 kW – does the engineer want to size the power conduit to provide median power, or maximum power?

Merlin Orchestration Execution

In this specification we will be using the [helics_cli](#) to execute each co-simulation run since this is a Monte Carlo simulation. This means that [helics_cli](#) will be executed multiple times with different [helics_cli](#) runner files.

Co-simulation Reproduction

Management of multiple iterations of the co-simulation can be done by setting the seed as a function of the number of brokers, where there will be one broker for each iteration. In Python 3.X:

```
import argparse

parser = argparse.ArgumentParser(description="EV simulator")
parser.add_argument(
    "--seed",
    type=int,
    default=0,
    help="The seed that will be used for our random distribution",
)
parser.add_argument("--port", type=int, default=-1, help="port of the HELICS broker")
```

(continues on next page)

(continued from previous page)

```
args = parser.parse_args()
np.random.seed(args.seed)
```

helics_cli in Merlin

Since we are using the `helics_cli` to manage and execute all the federates, we need to create these runner files for `helics_cli`. There is a provided python script called `make_samples_merlin.py` (see the `simple` subfolder in the code for the example) that will generate the runner file and a csv file that will be used in the study step. `helics_cli` will start each of these federates. In the Merlin spec, Merlin will be instructed to execute the `helics_cli` with all the generated `helics_cli` runner files.

Merlin Specification

Environment

In the Merlin spec we will instruct Merlin to execute N number of the Monte Carlo co-simulations. The number of samples is the number specified as the `N_SAMPLES` env variable in the env section of the Merlin spec.

```
env:
  variables:
    OUTPUT_PATH: ./UQ_EV_Study
    N_SAMPLES: 10
```

We set the output directory to `UQ_EV_Study`, this is where all the output files will be stored. Every co-simulation run executed by Merlin will have its own subdirectory in `./UQ_EV_Study`.

Merlin Step

Remember this step is for Merlin to setup all the files and data it needs to execute its jobs. In the Monte Carlo co-simulation there is a python script we created that will generate the `helics_cli` runner files that Merlin will use when it executes the `helics_cli`. The `make_samples_merlin.py` script (located under the `simple` subfolder of the advanced orchestration example code) will also output a csv file that Merlin will use. The csv file contains all the names of the runner files. Merlin will go down this list of file names and execute the `helics_cli` for each file name.

```
merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/simple/make_samples_merlin.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/Battery.py $(MERLIN_INFO)
        cp $(SPECROOT)/Charger.py $(MERLIN_INFO)
      file: $(MERLIN_INFO)/samples.csv
      column_labels: [FED]
```

The samples the get generated should look something like below. This is the runner file that `helics_cli` will use to start the co-simulation.

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker --federates=2 --loglevel=5 --type=tcpss --port 12345",
      "host": "broker",
      "name": "broker_0",
      "loglevel": 3
    },
    {
      "directory": ".",
      "exec": "python3 Battery.py --port 12345 --seed 1",
      "host": "broker",
      "name": "Battery",
      "loglevel": 3
    },
    {
      "directory": ".",
      "exec": "python3 Charger.py --port 12345",
      "host": "broker",
      "name": "Charger",
      "loglevel": 3
    }
  ],
  "name": "Generated by make samples"
}
```

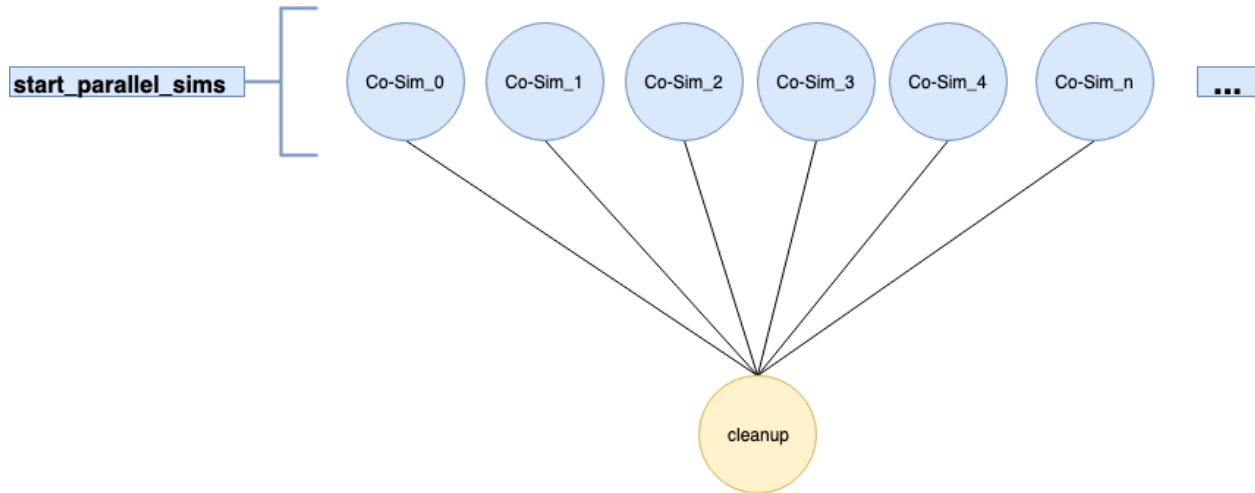
Once the samples have been created, we copy the 2 federates to the MERLIN_INFO directory.

Study Step

We have made it to the study step. This step will execute all 10 Monte Carlo co-simulations. There are 2 steps in the study step. The first step is the `start_parallel_sims` step. This step will use the `helics_cli` to execute each co-simulation. The second step, `cleanup` depends on the first step. Once `start_parallel_sims` is complete the `cleanup` step will remove any temporary data that is no longer needed.

```
study:
- name: start_parallel_sims
  description: Run a bunch of UQ sims in parallel
  run:
    cmd: |
      spack load helics
      helics run --path=$(MERLIN_INFO)/$(FED)
      echo "DONE"
- name: cleanup
  description: Clean up
  run:
    cmd: rm $(MERLIN_INFO)/samples.csv
    depends: [start_parallel_sims_*]
```

Below is what the DAG of the Merlin study will look like. Each of the Co-Sim_n bubbles represents the Monte Carlo simulation. Each co-sim runs in parallel with each other since there is no dependency on the output that each co-sim runs.



Questions and Help

Do you have questions about HELICS or need help?

1. Come to [office hours](#)!
2. Post on the [gitter](#)!
3. Place your question on the [github forum](#)!

Default HELICS setup:

Simulator int: external json
 Messages + Communication: endpoints
 + pub/sub
 Types of federates: python
 Broker hierarchies: 1

Let's learn about...

FEDERATE MESSAGE + COMMUNICATION CONFIGURATION

Simulator int: external json
Messages + Communications → pub/sub + multi-source inputs
 Types of federates: python
 Brokers: 1

Simulator int: external json
Messages + Communication → endpoints + queries
 Types of federates: python
 Brokers: 1

Simulator int: external json
Messages + Communication → python ns3
Types of federates → python python
 Brokers: 1

Simulator int: external json
Messages + Communication → pub/sub + ns3
Types of federates → python + ns3
 Brokers: 1

BROKERS

Simulator int: external json
 Messages + Communication: pub/sub
 Types of federates: python
Brokers → multiple core types

Simulator int: external json
 Messages + Communication: pub/sub
 Types of federates: python
Brokers → hierarchies

ORCHESTRATION

Simulator int: external json
 Messages + Communication: pub/sub
 Types of federates: python
Brokers → MERLIN for orchestration

MANY TYPES OF FEDERATES

Simulator int → external JSON
 Messages + Communication: pub/sub
Types of federates → python + GLD
 Broker hierarchies: 1

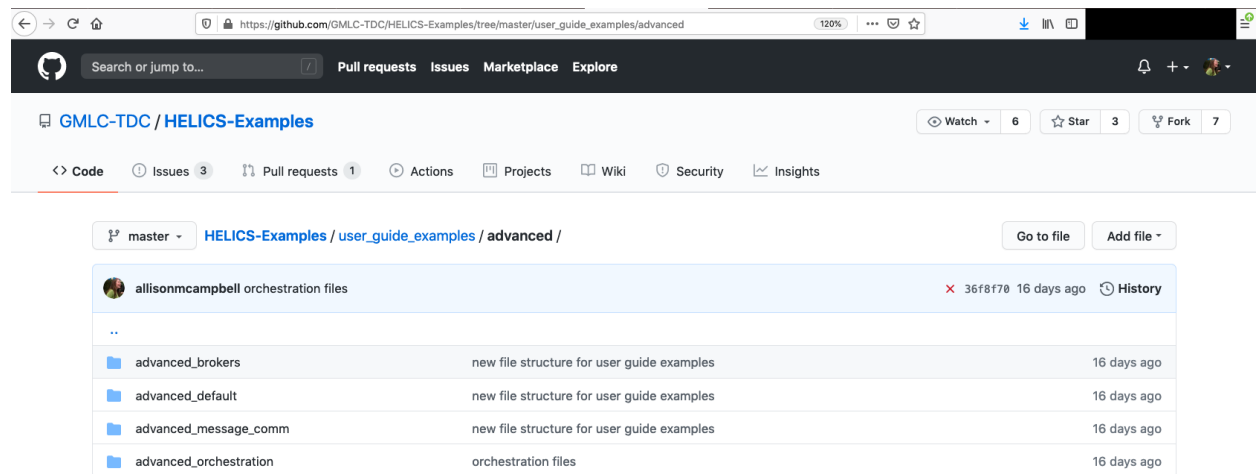
The Advanced Examples are modular, each building upon the *base example*. Users are encouraged to dive into a specific concept to build their HELICS knowledge once they have mastered the default setup in the *base example*.

This page describes the model for the Advanced Examples. This model is topically the same as the Fundamental Examples, but more complex in its execution. This allows the research question to become more nuanced and for the user to define new components to a HELICS co-simulation.

- *Where is the code?*
- *What is this Co-simulation doing?*
 - *Differences Compared to the Fundamental Examples*
 - * *HELICS Differences*
 - * *Research Question Complexity Differences*
- *HELICS Components*
 - *Federates with infinite time*
 - *Initial time requests and model initialization*

The code for the *Advanced examples* can be found in the HELICS-Examples repository on github. If you have issues navigating the examples, visit the HELICS [Gitter page](#) or the [user forum on GitHub](#).

TODO: UPDATE IMAGE



The Advanced Examples are similar in theme to the *Base Example* in that both are looking at power management for an EV charging garage. The implemented federates, however, are slightly more sophisticated and include a new centralized charging controller federate.

- **Battery.py** - Models a set of the EV batteries being charged. The EV is randomly assigned to support a particular charging level and receives an applied charging voltage based on that level. Using the applied voltage and the current SOC (initially randomly assigned), a charging current is calculated returned to the charger.
- **Charger.py** - Models a set of EV charging terminals all capable of supporting three defined charging levels: level 1, 2, and 3. Applies a charging voltage based on the charging terminal power rating and (imperfectly) measures the returned current. Based on this current, it estimates the SOC and sends that information to the controller. When commanded to terminate charging it removes the applied charging voltage.
- **Controller.py** - Receives periodic updates about the SOC of each charging vehicle and when they are considered close enough to full, command the charger to terminate charging.

Every time charging is terminated on an EV, a new EV to take its place is randomly assigned a supported charging level and initial SOC.

There are a few important distinctions between the Fundamental Examples and the Advanced Examples, which can be grouped into **HELICS** differences and **research question complexity** differences.

1. **Communication:** Both physical value exchanges and abstract information exchanges are modeled. The exchange of physical values takes place between the Battery and Charger federates (this was also introduced in a slimmed-down fashion in the *Fundamental Communication Example*). The message exchange (control signals, in this case) takes place between the Charger and Controller federates. For a further look at the difference between these two messaging mechanisms see our User Guide page on value federates and message federates.
2. **Timing:** The Controller federate has no regular update interval. The Controller works in pure abstract information and has no regular time steps to simulate. As such, it requests a maximum simulated time supported by HELICS (`HELICS_TIME_MAXTIME`) and makes sure it can be interrupted by setting `uninterruptable` to `false` in its configuration file. Any time a message comes in for the Controller, HELICS grants it a time, the Controller performs the required calculation, sends out a new control signal, and requests `HELICS_TIME_MAXTIME` again.

In the *Fundamental Base Example*, a similar research question is being addressed by this co-simulation analysis: estimate the **instantaneous power draw** from the EVs in the garage. And though you may have similar questions, there are several complicating changes in the new model:

1. A **third federate** (where previously there were only two) models responsibilities of a charger. The charger stops charging the battery by removing the charging voltage rather than the battery stopping the charging process. The Battery federate synthesizes an EV battery when the existing EV is considered fully charged.
2. The measurement of the charging current (used to calculate the actual charging power) has some **noise** built into it. This is modeled as random variation of the charging current in the federate itself and is a percentage of the charging current. The charging current decreases as the SOC of the battery increases leading to a noisier SOC estimate by the Charger federate at higher SOC. This results in the Controller tending to terminate charging prematurely as a single sample of the noisy charging current can lead to over-estimation of the SOC.
3. We can now model both **physics** and **measurement of physics**. There are two SOC values modeled in this co-simulation: the “actual” SOC of the battery modeled in the Battery federate and the estimate of the SOC as measured by the Charger federate. Both federates calculate the SOC in the same manner: use the effective resistive load of the battery, R , and a pre-defined relationship between R and SOC. You can see that both the Battery and Charger federates use the exact same relationship between SOC and effective R (SOC of zero is equivalent to an effective resistance of 8 ohms; SOC of 1 has an effective resistance of 150 ohms). Due to the noise in the charger current measurement, there is error built into its calculation of the SOC and therefore should be considered an estimate of the SOC.

This existence of two values for one property is not uncommon and is as much a feature as a bug. If this system were to be implemented in actual hardware, the only way that a charger would know the SOC of a battery would be through some kind of external **measurement**. And certainly there would be times where the charger would have even less information (such as the specific relationship between SOC and effective resistance) and would have to use historical data, heuristics, or smarter algorithms to know how to charge the battery effectively. Simulation allows us to use two separate models and thus independently model the actual SOC as known by the battery and the estimated SOC as calculated by the charger.

Since the decision to declare an EV fully charged has been abstracted away from the Charger and Battery (to the Controller), a slightly different procedure is used to disconnect a charged EV from the charger and replace it with a new one to be charged. In this advanced example, a mini-protocol has been designed and implemented:

1. The Charger receives a message from the Controller indicating the EV should be considered fully charged.
2. The Charger reduces the Charging voltage to zero volts and publishes it.
3. The Battery, detecting this change in charging voltage, infers that it is fully charged. The Battery federate instantiates a new EV with a battery at a random initial state of charge. The Battery federate also calculates a charging current of zero amps and publishes it.
4. The Charger federate, seeing a charging current of zero amps, infers a new EV has been set up to charge, randomly assigns one of three charging powers, and publishes this new charging voltage.

At this point the co-simulation proceeds as previously defined. The Battery uses its internal knowledge of the state-of-charge to define the charging current which the Charger uses to estimate the state-of-charge and sends on to the Controller. The Controller sends back a message to the Charger based on this state-of-charge estimate indicating whether the EV should continue to be charged or not.

The HELICS components introduced in the Fundamental Examples are extended in the Advanced Examples with additional discussion of timing and initialization of federates. These new components enter into the sequence as follows:

1. Register and Configure Federates
2. Initialization
3. Enter Execution Mode
4. Define Time Variables
5. Tell Controller federates to request `h.HELICS_TIME_MAXTIME`
6. Initiate Time Steps for the Time Loop
7. Send and Receive Communication between Federates
8. Finalize Co-simulation

Federates which are abstractions of reality (e.g., controllers) do not need regular time interval updates. These types of federates can be set up to request `HELICS_TIME_MAXTIME` (effectively infinite time) and only update when a new message arrives for it to process. This component is placed prior to the main time loop.

TODO: Get rid of `fake_max_time`

why do we divide `helics_time_maxtime` by 1000?

shouldn't we also show how this federate is configured? where is best for that?

```
hours = 24 * 7 # one week
total_interval = int(60 * 60 * hours)
grantedtime = 0
fake_max_time = int(h.HELICS_TIME_MAXTIME / 1000)
starttime = fake_max_time
logger.debug(f"Requesting initial time {starttime}")
grantedtime = h.helicsFederateRequestTime(fed, starttime)
logger.debug(f"Granted time {grantedtime}")
```

As in the *Base Example*, the EV batteries are assumed connected to the chargers at the beginning of the simulation and information exchange is initiated by the Charger federate sending the charging voltage to the Battery federate. In the Advanced Examples, this is a convenient choice as the charging voltage is constant and thus is never a function of the charging current. In a more realistic model, it's easy to imagine that the charger has an algorithm that adjusts the charging voltage based on the charging current to, say, ensure the battery is charged at a particular power level. In that case, **the dependency of the models is circular**; this is common component that needs to be addressed.

If the early time steps of the simulation are not as important (a model warm up period), then ensuring each federate has a default value it will provide when the input is null (and assuming the controller dynamics are not overly aggressive) will allow the models to bootstrap and through several iterations reach a consistent state. If this is not the case then HELICS does have a provision for getting models into a consistent state prior to the start of execution: initialization mode. **TODO: link to documentation or example on initialization mode.** This mode allows for this same iteration between models with no simulated time passing. It is the responsibility of the modeler to make sure there is a method to reach and detect convergence of the models and when such conditions are met, enter execution mode as would normally be done.

todo: add examples for where this is inserted in the code

Using the *Advanced Default Example* as the starting point, the following examples have also been constructed:

- **Multi-Source Inputs** - Demonstration of use and configuration of a multi-sourced input value handle.
- **Queries** - Demonstration of the use of queries for dynamic federate configuration.
- **Multiple Brokers**
 - **Connecting Multiple Core Types (Multi-Protocol Broker)** - Demonstration of how to configure a multi-protocol broker
 - **Broker Hierarchies** - Purpose of broker hierarchies and how to configure a HELICS co-simulation to implement one.
 - **Simultaneous co-simulations** - Demonstration of how to run multiple independent federations simultaneously on a single compute node.
- **Orchestration Tool (Merlin)** Demonstration of using [Merlin](#) to handle situations where a HELICS co-simulation is just one step in an automated analysis process (*e.g.* uncertainty quantification) or where assistance is needed deploying a large co-simulation in an HPC environment.

1.5 Support

If you're having trouble with understanding what HELICS does, how to use HELICS, getting a specific feature to work, or trouble-shooting some aspect of a HELICS-based co-simulation; there are a variety of support mechanisms we offer.

- **Documentation** - Though you're already here at the documentation site, the following pages may be of particular interest:
 - **Installation Guide** - Help with getting HELICS installed with the correct options for your particular use case
 - **User Guide** - Comprehensive guide to understanding and using HELICS covering both the *fundamentals* as well as more *advanced topics*.
 - **API reference** - HELICS is written in C++ with many supported language bindings. The C++ Doxygen is the most comprehensive reference for the HELICS API
- **User Forum** - A space where users can discuss their HELICS use cases and get support on how to use HELICS.
- **Bug Reports/Support** - If HELICS doesn't appear to be working as expected, this is the place to file a bug report and get some help.
- **Examples, Use Cases, and Tutorials** - Over the years there have been a number of different public examples, use cases, and demonstrations of HELICS. These may be useful to show how particular features work or as a starting point for your use of HELICS.
 - [NREL March 2020 In-house Tutorial](#)
 - [HELICS Examples Github repository](#)
 - [HELICS Fundamental Tutorials](#)
 - [Demonstration Use Cases](#) - Fully-formed co-simulation use cases using a variety of simulators and HELICS features.
- **YouTube Channel** - HELICS developers have created a number of short lectures and tutorials about how HELICS works and how to use it.
- **Gitter** - Live chat with developers (when they're logged in) for quick fix support.
- **Virtual Office Hours** - HELICS developers have periodic open office hours where anybody can come and get hands-on help with understanding HELICS better or getting a particular feature to work. Contact helics-team@helics.org for an invitation.

DEVELOPER GUIDE

2.1 Style Guide

The goal of the style guide is to describe in detail naming conventions for developing HELICS. Style conventions are encapsulated in the .clang_format files in the project.

We have an EditorConfig file that has basic formatting rules code editors and IDEs can use. See <https://editorconfig.org/> for how to setup support in your preferred editor or IDE.

2.1.1 Naming Conventions

1. All functions should be camelCase

```
PublicationID registerGlobalPublication (const std::string &name, const std::string &type, const std::string &units = "");
```

EXCEPTION: when the functionality matches a function defined in the standard library e.g. to_string()

2. All classes should be PascalCase

```
class ValueFederate : public virtual Federate
{
public:
    ValueFederate (const FederateInfo &fi);
}
```

3. class methods should be camelCase

```
Publication &registerGlobalPublication (const std::string &name, const std::string &type, const std::string &units = "");
```

Exceptions: functions that match standard library functions e.g. to_string()

4. Enumeration names should be PascalCase. Enumeration values should be CAPITAL_SNAKE_CASE

```
/* Type definitions */
typedef enum {
    HELICS_OK,
    HELICS_DISCARD,
    HELICS_WARNING,
    HELICS_ERROR,
} HelicsStatus;
```

5. Constants and macros should CAPITAL_SNAKE_CASE
6. Variable names: local variable names should be camelCase member variable names should be mPascalCase static const members should be CAPITAL_SNAKE_CASE function input names should be camelCase index variables can be camelCase or ii, jj, kk, mm, nn or similar if appropriate global variables should gPascalCase
7. All C++ functions and types should be contained in the helics namespace with subnamespaces used as appropriate

```
namespace helics
{
    ...
} // namespace helics
```

8. C interface functions should begin with helicsXXXX

```
HelicsBool helicsBrokerIsConnected (HelicsBroker broker);
```

9. C interface function should be of the format helics{Class}{Action} or helics{Action} if no class is appropriate

```
HelicsBool helicsBrokerIsConnected (HelicsBroker broker);

const char *helicsGetVersion ();
```

10. All cmake commands (those defined in cmake itself) should be lower case

```
if as opposed to IF
install vs INSTALL
```

1. Public interface functions should be documented consistent with Doxygen style comments non public ones should be documented as well with doxygen but we are a ways from that goal

```
/** get an identifier for the core
    @param core the core to query
    @return a string with the identifier of the core
 */
HELICS_EXPORT const char *helicsCoreGetIdentifier (HelicsCore core);
```

1. File names should match class names if possible
2. All user-facing options (*e.g.* log_level) should be expressed to the user as a single word or phrase using nocase, camelCase, and snake_case. Do not use more than one synonymous term for the same option; that is, do not define a single option that is expressed to the user as both best_ice_cream_flavor and most_popular_ice_cream_flavor.

Exception: when defining the command line options, the command-line parser already handles underscores and casing so there is no need to define all three cases for that parser.

2.2 Generating SWIG extension

MATLAB

For the MATLAB extension, you need a special version of SWIG. Get it [here](#).

```
git clone https://github.com/jaeandersson/swig
cd swig
./configure --prefix=/Users/${whoami}/local/swig-matlab/ && make -j8 && make install
```

The matlab interface can be built using `HELICS_BUILD_MATLAB_INTERFACE` in the CMake build of HELICS. This will use a MATLAB installation to build the interface. See installation

Octave

Octave is a free program that works similarly to MATLAB Building the octave interface requires swig, and currently will work with Octave 4.0 through 4.2. 4.4 is not currently supported by SWIG unless you build from the current master branch of the swig repo and use that version. The next release of swig will likely support it. It does work on windows, though the actual generation is not fully operational for unknown reasons and will be investigated at some point. A `mkhelicsOCTFile.m` is generated in the build directory this needs to be executed in octave, then a `helics.oct` file should be generated, the `libHelicsShared.dll` needs to be copied along with the `libzmq.dll` files Once this is done the library can be loaded by calling `helics`. On linux this build step is done for you with `HELICS_BUILD_OCTAVE_INTERFACE`.

C# A C# interface can be generated using swig and enabling `HELICS_BUILD_CSHARP_INTERFACE` in the CMake. The support is partial; it builds and can be run but not all the functions are completely usable and it hasn't been fully tested.

Java A JAVA interface can be generated using swig and enabling `HELICS_BUILD_JAVA_INTERFACE` in the CMake. This interface is tested regularly as part of the CI test system.

2.3 Run tests

TODO: Add something or remove this file

2.4 Generating Documentation

The documentation requires Pandoc to convert from Markdown to RST.

You will need the following Python packages.

```
pip install sphinx
pip install ghp-import
pip install breathe
pip install sphinx_rtd_theme
pip install nbsphinx
pip install sphinxcontrib-pandoc-markdown
```

You will also need doxygen.

You can then type `make doxygen html` to create the documentation locally.

If you don't have Pandoc, you can install it using `conda`.

```
conda install pandoc
```

If you are unable to install pandoc, you may be able to generate some of the documentation if you install the following.

```
pip install recommonmark
```

2.5 HELICS Benchmarks

The HELICS repository has a few benchmarks that are intended to test various aspects of the code and record performance over time

2.5.1 Baseline benchmarks

These benchmarks run on a single machine using Google Benchmarks and are intended to test various aspects of HELICS over a range of spaces applicable to a single machine.

ActionMessage

Micro-benchmarks to test some operations concerning the serialization of the underlying message structure in HELICS

Conversion

Micro-benchmarks to test the serialization and deserialization of common data types in HELICS

2.5.2 Simulation Benchmarks

Echo

A set of federates representing a hub and spoke model of communication for value based interfaces

Echo_c

A set of federates representing a hub and spoke model of communication for value based interfaces using the C shared library.

Echo Message

A set of federates representing a hub and spoke model of communication for message based interfaces

Filter

A variant of the Echo message test that add filters to the messages

Ring Benchmark

A ring like structure that passes a value token around a bunch of times

Ring Message Benchmark

A ring like structure that passes a message token around a bunch of times.

Timing Benchmark

Similar to echo but doesn't actually send any data just pure test of the timing messages

2.5.3 Message Benchmarks

Benchmarks testing various aspects of the messaging structure in HELICS

MessageLookup

Benchmarks sends messages to random federates, varying the total number of interfaces and federates.

MessageSend

Sending messages between 2 federates varying the message size and count per timing loop.

2.5.4 Standardized Tests**PHold**

A standard PHOLD benchmark varying the number of federates.

2.5.5 Multinode Benchmarks

Some of the benchmarks above have multinode variants. These benchmarks will have a standalone binary for the federate used in the benchmark that can be run on each node. Any multinode benchmark run will require some setup to make it launch in your particular environment and knowing the basics for the job scheduler on your cluster will be very helpful.

Any sbatch files for multinode benchmark runs in the repository are setup for running in the pdebug queue on LC's Quartz cluster. They are unlikely to work as is on other clusters, however they should work as a starting point for other clusters using slurm. The minimum changes required are likely to involve setting the queue/partition correctly and ensuring the right bank/account for charging CPU time is used.

2.6 Description of the different continuous integration test setups running on the CI servers

There are 5 CI servers that are running along with a couple additional checks Travis, Appveyor, Circle-CI, Azure and Drone.

2.6.1 Travis-CI Tests

Travis-ci runs many of the primary checks In 3 different stages

Push Tests

Push tests run on all pushes to any branch in the main repository, there are 4 tests that run regularly

- GCC 6: Test the GCC 6.0 compiler and the CI labeled Tests BOOST 1.61, SWIG, MPI
- Clang 5: Test the clang compiler and run the CI labeled Tests, along with Java interface generation and Tests Using C++17
- GCC 4.9: Test the oldest supported compiler in GCC, Test the included interface files(SWIG OFF) for Java, and test a packaging build. The main tests are disabled, BOOST 1.61
- XCode 10.2: Test a recent XCode compiler with the Shared API library tests

PR tests and develop branch Tests

Pull request tests run on every pull request to develop or main. In addition to the previous 4 tests 2 additional tests are run.

- Clang 3.6: which is the oldest fully supported clang compiler, with boost 1.58 (Build only)
- XCode 10.2

Daily Builds on develop

On the develop branch a few additional tests are run on a daily basis. These will run an extended set of tests or things like valgrind or clang-sanitizers. The previous tests are run with an extended set of tests and a few additional tests are run

- gcc 6.0 valgrind, interface disabled
- gcc 6.0 Code Coverage, MPI, interfaces disabled
- gcc 6.0 ZMQ subproject cmake 3.11
- Mingw test building on the Mingw platform
- Xcode 9.4 which is the oldest fully supported Xcode version (not for PRs to develop)

2.6.2 Appveyor tests

- MSVC 2015 CMake 3.13, and JAVA builds

2.6.3 Azure tests

PRs and commits to the main and develop branches that pass the tests on Travis will trigger builds on Azure for several other HELICS related repositories (such as HELICS-Examples). The result of the builds for those repositories will be reported as a comment on the PR (if any) that triggered the build.

On the Primary HELICS repository there are 4 Azure builds:

- MSVC2015 64bit Build and test, chocolatey swig/boost
- MSVC2017 32bit Build and test
- MSVC2017 64bit Build and test with Java
- MSVC2019 64bit Build and test with Java

2.6.4 Circle CI

All PR's and branches trigger a set of builds using Docker images on Circle-CI.

- Octave tests - tests the Octave interface and runs some tests
- Clang-MSAN - runs the clang memory sanitizer
- Clang-ASAN - runs the clang address sanitizer and undefined behavior sanitizer
- Clang-TSAN - runs the clang thread sanitizer
- install1 - build and install and link with the C shared library, C++ shared library, C++98 library and C++ apps library, and run some tests linking to the installed libraries
- install2 - build and install and link with the C shared library, and C++98 library only and run some tests linking with the installed library

Benchmark tests

Circle ci also runs a benchmark test that runs every couple days. Eventually this will form the basis of benchmark regression test.

2.6.5 Drone

- 64 bit and 32 bit builds on ARM processors

2.6.6 Cirrus CI

- FreeBSD 12.1 build

2.6.7 Read the docs

- Build the docs for the website and test on every commit

2.6.8 Codacy

There are some static analysis checks run with Codacy. While it is watched it is not always required to pass.

2.7 (Planned) CI/CD Infrastructure

This section documents the services used by HELICS.

2.7.1 Continuous Integration

The GMLC-TDC/HELICS repository uses Azure Pipelines to test the main platforms and compilers we support, with Drone Cloud and Cirrus CI for testing some less common platforms, and CircleCI for running tests using sanitizers. Nightly release builds are run on GitHub Actions.

All of the builds on Linux use Docker containers. This has a number of advantages:

- The build environment is consistent, making changes to the underlying image easy
- Tools don't need reinstalling for each build
- The tests can be run locally on a developer machine in the same environment
- Old build environments can be used for binary compatibility, such as CentOS 6 for releases.

To avoid long build queues, commits to `main` and `develop` will be tested with more configurations, while PRs and commits to other branches may be tested on a smaller set of platforms. PRs and commits to `main` should run a full set of tests on all supported platforms to help ensure that it is always ready for a new release. Jobs that run for PRs and all commits are marked with `[commit]`, while jobs marked with `[daily]` are only run daily on the `develop` branch. A tag of `[main]` indicates that the job is only run for commits and PRs to `main`.

If more extensive testing is needed for a commit or branch, certain keywords can be included in the commit message to trigger additional builds for commits or in the branch name to trigger them for a particular branch. At some point in the future, the community server may provide ways to trigger extra builds in other circumstances.

Unless otherwise mentioned, jobs will typically generate the Java interface, run the CI/nightly test suite, and run some brief packaging tests.

Linux

For Linux, typically the jobs run will be the latest gcc and clang releases, and the oldest gcc and clang releases supported by HELICS. Unless indicated, they will use a system install of ZeroMQ, build all supported core types, and use a fairly recent version of dependencies.

- GCC 7.0 on Ubuntu 18.04 using default apt-get versions of dependencies
- Clang 5.0 using minimum supported version of all dependencies
- GCC 10 [commit]
- Clang 10 [commit]
- GCC 10 with ZeroMQ as a subproject [daily]
- CentOS 5 or 6 (same image as for releases)

For auto-generated swig PRs, there is a special build that will run to test a build using the pre-generated swig interface files.

Windows

Most of the Windows tests run frequently are using MSVC, though MinGW, MSYS, and Cygwin jobs are run daily. The MINGW/MSYS builds can be triggered by including `mingw`, `msys`, and/or `cygwin` in a commit message or branch name.

- MSVC2017 32bit Build and test
- MSVC2017 64bit Build and test with Java
- MSVC2019 64bit Build and test with Java [commit]
- MinGW [daily]
- MSYS [daily]
- Cygwin 32-bit [daily]

macOS

macOS builds and tests tend to take a long time to run, so there are only a few of them, and they only run on PRs unless `xcode10` or `xcode11` is included in a commit message or branch name. In general, the XCode jobs will be the most recent major XCode version and the oldest XCode version released in about the past 3 years that's supported by Apple (or is available on the Azure macOS images). Apple tends to be pretty good at getting people to upgrade and dropping support for older releases.

- XCode 10.1
- XCode 11

ARM

ARM (32-bit) and aarch64 builds run on Drone Cloud with the regular CI tests for all pull requests and commits. These builds use the latest Alpine Linux docker image for the builder, which uses musl libc instead of glibc.

FreeBSD

A FreeBSD 12.1 build runs on Cirrus CI with short system tests for all pull requests and commits.

Valgrind [daily]

A build with valgrind is run daily to catch memory management and threading bugs.

Code Coverage [daily]

A daily job runs tests on Linux to gather code coverage results that are uploaded to codecov.

Sanitizers

Sanitizers and some install tests are run in Docker containers on Circle CI for all commits and PRs.

- Octave tests - tests the Octave interface and runs some tests
- Clang-MSAN - runs the clang memory sanitizer
- Clang-ASAN - runs the clang address sanitizer and undefined behavior sanitizer
- Clang-TSAN - runs the clang thread sanitizer
- install1 - build and install and link with the C shared library, C++ shared library, C++98 library and C++ apps library, and run some tests linking to the installed libraries
- install2 - build and install and link with the C shared library, and C++98 library only and run some tests linking with the installed library

Documentation

ReadTheDocs is used to build and host the HELICS documentation.

2.7.2 Static Analysis, Linting, and Automated Code Maintenance

GitHub Actions is the main service used for running static analysis and linting tools, and automating some code related maintenance tasks.

pre-commit

The pre-commit workflow runs linters and formatting tasks for non-C++ code, spell checking for docs and comments, and opens a PR with any fixes.

Updating generated interface files and docs

The swig-gen workflow updates the swig generated interface files for the most commonly used language bindings for HELICS. In the near future it will also perform some documentation generating tasks to keep documentation on different language interfaces up-to-date.

2.7.3 Automated Releases

GitHub Actions is used to build release packages when a new release is created on GitHub. It also triggers jobs that start the process of updating HELICS in several package repositories.

2.7.4 Multinode Tests and Benchmarks

A cluster on AWS is used to run multinode tests and benchmarks to help detect performance regressions.

Circle CI runs benchmarks every couple of days on a single machine, which serves as the basis for a benchmark performance regression test.

2.7.5 Community Server

A small AWS instance is used to perform various orchestration tasks related to releases, and controlling the cluster used for multinode regression tests. It also runs bots that use GitHub events for providing some services such as repository maintenance tasks and monitoring the automated release process.

2.8 Porting Guide: HELICS 2 to 3

Since HELICS 3 is a major version update, there are some breaking changes to the API for developers. This guide will try to track what breaking changes are made to the API, and what developers should use instead when updating from HELICS 2.x to 3.

2.8.1 Dependency changes

Support for some older compilers and dependencies have been removed. The new minimum version are:

- C++17 compatible-compiler (minimums: GCC 7.0, Clang 5.0, MSVC 2017 15.7, XCode 10, ICC 19)
- CMake 3.10+ (if using clang with libc++, use 3.18+)
- ZeroMQ 4.2+
- Boost 1.65.1+ (if building with Boost enabled)

2.8.2 Code changes

Changes that will require changing code are listed below based on the interface API used. A list of known PRs that made breaking changes is also provided.

PRs with breaking changes

- [#1363](#)
- [#1952](#)
- [#1907](#)
- [#1856](#)
- [#1731](#)
- [#1727](#)
- [#1680](#)
- [#1679](#)
- [#1677](#)
- [#1572](#)
- [#1580](#)

Application API (C++17)

- `Federate::error(int errorcode)` and `Federate::error(int errorcode, const std::string& message)` were removed, use `localError` instead (or `globalError` to stop the entire simulation). Changed in [#1363](#).
- `ValueFederate::publishString` and `ValueFederate::publishDouble` have been removed, please use the `Publication` interface `publish` methods which take a wide variety of types
- The interface object headers are now included by default when including the corresponding federate

Command line interfaces

The numerical value corresponding with the log levels have changed. As such entering numerical values for log levels is no longer supported (it will be again someday). For now please use the text values “none(-1)”, “no_print(-1)”, “error(0)”, “warning(1)”, “summary(2)”, “connections(3)”, “interfaces(4)”, “timing(5)”, “data(6)”, “debug(6)”, “trace(7)”. The previous values are shown in parenthesis. The new numerical values are subject to revision in a later release so are not considered stable at the moment and are not currently accepted as valid values for command line or config files.

C Shared API

- Only 1 header is now used `#include <helics/helics.h>` for all uses of the C shared library in C/C++ code – no other headers are needed, the other headers are no longer available. [#1727](#)
- Removed `helics_message` struct – call functions to set fields instead. `helicsEndpointGetMessage` and `helicsFederateGetMessage` returning this struct were removed – call functions to get field values instead. Changed in [#1363](#).
- `helics_message_object` typedef was renamed to `HelicsMessage` in `api-data.h`; in `MessageFederate.h` and `helicsCallbacks.h` all `helics_message_object` arguments and return types are now `HelicsMessage`. Changed in [#1363](#).
- Renamed `helicsEndpointSendMessageObject` to `helicsEndpointSendMessage`, `helicsSendMessageObjectZeroCopy` to `helicsSendMessageZeroCopy`, `helicsEndpointGetMessageObject` to `helicsEndpointGetMessage`, `helicsEndpointCreateMessageObject` to `helicsEndpointCreateMessage`, `helicsFederateGetMessageObject` to `helicsFederateGetMessage`, and `helicsFederateCreateMessageObject` to `helicsFederateCreateMessage`. Changed in [#1363](#).
- The send data API has changed to make the usage clearer and reflect the addition of targeted endpoints. New methods are `helicsEndpointSendBytes`, `helicsEndpointSendBytesTo`, `helicsEndpointSendBytesAt`, `helicsEndpointSendBytesToAt`. This reflects usage to send a raw byte packet to the targeted destination or a user specified one, and at the current granted time or a user specified time in the simulation future. The C++98 API was changed accordingly. The order of fields has also changed for consistency [#1677](#)
- Removed `helicsEndpointClearMessages` – it did nothing, `helicsFederateClearMessages` or `helicsMessageFree` should be used instead. Changed in [#1363](#).
- All constants such as flags and properties are now `CAPITAL_SNAKE_CASE` [#1731](#)
- All structures are now CamelCase, though the old form will be available in helics3 though will be deprecated at some point. [#1731](#) [#1580](#)
- `helicsPublicationGetKey` renamed to `helicsPublicationGetName` [#1856](#)
- Recommended to change `helicsFederateFinalize` to `helicsFederateDisconnect`, the finalize method is still in place but will be deprecated in a future release. [#1952](#).
- `helicsMessageGetFlag` renamed to `helicsMessageGetFlagOption` for better symmetry [#1680](#)
- `helics<*>PendingMessages` moved `helics<*>PendingMessageCount` [#1679](#)

C++98 API (wrapper around the C Shared API)

- Removed the `helics_message` struct, and renamed `helics_message_object` to `HelicsMessage`. Direct setting of struct fields should be done through API functions instead. This affects a few functions in the `Message` class in `Endpoint.hpp`; the explicit constructor and `release()` methods now take `HelicsMessage` arguments, and operator `helics_message_object()` becomes operator `HelicsMessage()`. Changed in [#1363](#).

Queries

- Queries now return valid JSON except for `global_value` queries. Any code parsing the query return value will need to be adjusted. Error codes are reported back as HTML error codes and a message.

Libraries

- The C based shared library is now `libhelics.dll` or the appropriate extension based on the OS [#1727](#) [#1572](#)
- The C++ shared library is now `libhelicscpp.[dll\so\dylib]` [#1727](#) [#1572](#)
- The apps library is now `libhelicscpp-apps.[dll\so\dylib]` [#1727](#) [#1572](#)

CMake

- All HELICS CMake variables now start with `HELICS_` [#1907](#)
- Projects using HELICS as a subproject or linking with the CMake targets should use `HELICS::helics` for the C API, `HELICS::helicscpp` for the C++ shared library, `HELICS::helicscpp98` for the C++98 API, and `HELICS::helicscpp-apps` for the apps library. If you are linking with the static libraries you should know enough to be able to figure out what to do, otherwise it is not recommended.

2.9 Public API

This file defines what is included in what is considered the stable user API for HELICS.

This API will be backwards code compatible through major version numbers, though functions may be marked deprecated between minor version numbers. Functions in any other header will not be considered in versioning decisions. If other headers become commonly used we will take that into consideration at a later time. Anything marked private is subject to change and most things marked protected can change as well though somewhat more consideration will be given in versioning.

The public API includes the following

- Application API headers
 - `CombinationFederate.hpp`
 - `Publications.hpp`
 - `Subscriptions.hpp`
 - `Endpoints.hpp`
 - `Filters.hpp`
 - `Federate.hpp`
 - `helicsTypes.hpp`
 - `data_view.hpp`
 - `MessageFederate.hpp`
 - `MessageOperators.hpp`
 - `ValueConverter.hpp`
 - `ValueFederate.hpp`

- HelicsPrimaryTypes.hpp
- queryFunctions.hpp
- FederateInfo.hpp
- Inputs.hpp
- BrokerApp.hpp
- CoreApp.hpp
- timeOperations.hpp
- typeOperations.hpp
- Exceptions: Vector subscriptions, and vector inputs are subject to change. The queries to retrieve JSON may update the format of the returned JSON in the future. A general note on queries. The data returned via queries is subject to change, in general queries will not be removed, but if a need arises the data structure may change at minor revision numbers.
- Core library headers
 - Core.hpp
 - Broker.hpp
 - core-exceptions.hpp
 - core-data.hpp
 - CoreFederateInfo.hpp
 - helicsVersion.hpp
 - federate_id.hpp
 - helics_definitions.hpp
 - NOTE: core headers in the public API are headers that need to be available for the Application API public headers. The core API can be used more directly with static linking but applications are generally recommended to use the application API or other higher level API's
- C Shared Library headers (c) All C library operations are merged into a single header helics.h A helics_api.h header is available for generating interfaces which strips out import declarations and comments. The C shared library API is the primary driver of versioning and changes to that will be considered in all versioning decisions.
- App Library
 - Player.hpp
 - Recorder.hpp
 - Echo.hpp
 - Source.hpp
 - Tracer.hpp
 - Clone.hpp
 - helicsApp.hpp
 - BrokerApp.hpp (aliased to application_api version)
 - CoreApp.hpp (aliased to application_api version)
- Exceptions: The vector subscription Objects, and vector Input objects are subject to change.

- C++98 Library *All headers are mostly stable. Though we reserve the ability to make changes to make them better match the main C++ API.*

In the installed folder are some additional headers from third party libraries (CLI11, utilities), we will try to make sure these are compatible in the features used in the HELICS API, though changes in other aspects of those libraries will not be considered in HELICS versioning, this caveat includes anything in the `helics/external` and `helics/utilities` directories. Only changes which impact the signatures defined above will factor into versioning decisions. You are free to use them but they are not guaranteed to be backwards compatible on version changes.

2.10 RoadMap

This document contains tentative plans for changes and improvements of note in upcoming versions of the HELICS library. All dates are approximate and subject to change, but this is a snapshot of the current planning thoughts. See the [projects](#) for additional details

2.10.1 [2.8] ~ 2021-08-31

This will be the last of the 2.X series releases, there will likely be at least one patch release after this before fully moving to 3.0

- Internal text based (probably JSON) message format option for general backwards compatibility
- Function deprecations to prepare people to move to 3.0

2.10.2 [3.1] ~ 2021-09-28

Mostly things that didn't quite make it into the 3.0 release and a number of bug fixes that come from transitioning to HELICS 3.0.

- Full Dynamic Federation support
- Single thread cores (partial at release)
- Plugin architecture for user defined cores
- SSL capable core via plugin
- Full xSDK compatibility
- Much more general debugging support
- Performance improvements
- Bug fixes, tests and tuning related to HELICS 3 increased use.
- Separate out matlab HELICS interface in matlab oriented way vs current swig build

2.11 HELICS Type Conversions

HELICS has the ability to convert data between different types. In the C api methods are available to send and receive data as strings, integers, doubles, boolean, times, char, complex, vector of doubles, and named points.

2.11.1 Available types

The specification of publication allows setting the publication to one of following enum values:

```
HELICS_DATA_TYPE_UNKNOWN = -1,
/** a sequence of characters*/
HELICS_DATA_TYPE_STRING = 0,
/** a double precision floating point number*/
HELICS_DATA_TYPE_DOUBLE = 1,
/** a 64 bit integer*/
HELICS_DATA_TYPE_INT = 2,
/** a pair of doubles representing a complex number*/
HELICS_DATA_TYPE_COMPLEX = 3,
/** an array of doubles*/
HELICS_DATA_TYPE_VECTOR = 4,
/** a complex vector object*/
HELICS_DATA_TYPE_COMPLEX_VECTOR = 5,
/** a named point consisting of a string and a double*/
HELICS_DATA_TYPE_NAMED_POINT = 6,
/** a boolean data type*/
HELICS_DATA_TYPE_BOOLEAN = 7,
/** time data type*/
HELICS_DATA_TYPE_TIME = 8,
/** raw data type*/
HELICS_DATA_TYPE_RAW = 25,
/** type converts to a valid json string*/
HELICS_DATA_TYPE_JSON = 30,
/** the data type can change*/
HELICS_DATA_TYPE_MULTI = 33,
/** open type that can be anything*/
HELICS_DATA_TYPE_ANY = 25262
```

When this data is received it can be received as any of the available types, thus there are no restrictions on which function is used based on the data that was sent. That being said not all conversions are lossless. The following image shows which conversions are lossless through round trip operations. On the sending side the same is also true in that any of the setValue methods may be used regardless of what the actual transmission data type is set to.

For the remaining conversions the loss is typically numerical or comes with conditions. For example converting a vector to a complex number you can think of a complex number as a two element vector. Thus for 1 or 2 element vectors the conversion will be lossless. For 3 or more element vectors the remaining elements are discarded. For conversion of any vector to a single value, double, or integral the value is collapsed using vector normalization. Thus for single element vectors or real valued complex numbers the result is equivalent, whereas for others there is a reduction in information, which may be desired in some cases. Bool values get reduced to false if the numerical value is anything but 0, and true otherwise.

Conversion from strings to numeric values assume the string encodes a number, otherwise it results in an invalid value.

Invalid Values

Invalid values are returned if there is no value or the conversion is invalid for some reason, typically string to numeric value failures, in most cases an empty value equivalent is returned.

- INT -> -9,223,372,036,854,775,808 (`numeric_limits<std::int64_t>::min()`)
- DOUBLE -> -1e49
- COMPLEX -> {-1e49,0.0}
- VECTOR -> {}
- COMPLEX_VECTOR -> {}
- STRING -> ""
- NAMED_POINT -> {"", NaN}
- BOOL -> false
- TIME -> `Time::minVal() = Time(numeric_limits<std::int64_t>::min())`

2.11.2 How types are used in HELICS

Interface specification

For Value based interfaces types are used in a number of locations. For publications the most important part is specifying the type of the publication. This determines the type of data that gets transmitted. It can be a specific known type, a custom type string that is user defined, or an any type.

Inputs or subscriptions may also optionally specify a type as well. The use of this is for information to other federates, and for some type compatibility checking. It defaults to an “any” type so if nothing is specified it will match to any publication type.

Data publication and extraction

As mentioned before when using any of the `publication` methods the extraction method call is not limited based on the type given in the interface specification. What the publication type does is define a conversion if necessary. The same is true on the output.

`SetValueType -> PublicationType -> GetValueType`

Thus there are always two conversions occurring in the data translation pipeline. The first from the `setValue` to the type specified in the publication registration. The second is from the publish transmission type to the value requested in a respective `getValue` method.

2.11.3 Data Representation

`int` and `double` are base level types in C++. `complex` is two doubles, `vector` and `complex_vector` are variable length arrays of doubles. `named_point` is `string` and `double`. `String` is a variable length sequence of 8 bit values. `char` is a string of length 1, `boolean` is a single char either 0 or 1. `Time` is a representation of internal HELICS time as a 64 bit integer (most commonly number of nanoseconds). The `JSON` type is a string compatible with JSON with two fields “type” and “value”. The custom type is a variable length sequence of bytes which HELICS will not attempt to convert, it transmits a sequence of bytes and it would be up to the user to define any conversions or endianness concerns.

2.11.4 Data conversions

There are defined conversions from all known available types to all others.

Conversion from Double

- INT -> trunc(val)
- DOUBLE -> val
- COMPLEX -> val+0j
- VECTOR -> [val]
- COMPLEX_VECTOR -> [val+0j]
- NAMED_POINT -> {"value", val}
- STRING -> string representation such that all required bits are included
- BOOL -> (val!=0)?"1":"0"

Conversion from INT

- INT -> val
- DOUBLE -> val¹
- COMPLEX -> val+0j
- VECTOR -> [val]
- COMPLEX_VECTOR -> [val+0j]
- NAMED_POINT -> {"value", val}²
- STRING -> std::to_string(val)
- BOOL -> (val!=0)?"1":"0"

Conversion from String

- INT -> getIntFromString(val)
- DOUBLE -> getDoubleFromString(val)
- COMPLEX -> getComplexFromString(val)
- VECTOR -> helicsGetVector(val)
- COMPLEX_VECTOR -> helicsGetComplexVector(val)
- NAMED_POINT -> {val, NaN}
- STRING -> val
- BOOL -> (helicsBoolValue(val))?"1":"0"

¹ conversion to double is lossless only if the value actually fits in a double mantissa value.

² for a named point conversion, if the value doesn't fit in double the string translation is placed in the string field and a NaN value in the value segment to ensure no data loss.

helicsGetComplexVector

This method will read a vector of numbers from a string in either JSON format, or `c[X1,X2,...XN]` where `XN` is a complex number of the format `R+Ij`. It can also interpret `v[X1,X2,..., XN]` in which case the values are assumed to be alternating real and imaginary values. If the string is a single value either real or complex it is placed in a vector of length 1.

helicsGetVector

This function is similar to `helicsGetComplexVector` with distinction that string like `v[X1,X2,..., XN]` are all assumed separate values, and complex vectors are generated as alternating real and imaginary values.

getIntFromString

Converts a string into 64 bit integer. If the string has properties of a double or vector it will truncate the values from `getDoubleFromString`.

getDoubleFromString

Converts a string into a double value, a complex, or a vector of real or complex numbers. If the vector is more than a single element the output is the vector norm of the vector. If the string is not convertible the `invalid_double` is returned (-1e49).

helicsGetComplexFromString

Similar to `getDoubleFromString` in conversion of vectors. It will convert most representations of complex number patterns using a trailing `i` or `j` for the imaginary component and assumes the imaginary component is last. The real component can be omitted if not present, for example `4.7+2.7j` or `99.453i`

helicsBoolValue

`"0", "00", "0000", "0", "false", "f", "F", "FALSE", "N", "no", "n", "OFF", "off", "", "disable", "disabled"` all return false, everything else returns true.

Conversion from vector double

- INT -> `trunc(vectorNorm(val))`²
- DOUBLE -> `vectorNorm(val)`
- COMPLEX -> `val[0]+val[1]j`
- VECTOR -> `val`
- COMPLEX_VECTOR -> `[val[0],val[1],...,val[N]]`
- STRING -> `vectorString(val)`^{Page 191, 2}
- NAMED_POINT -> `{vectorString(val), NaN}`³

³ if the vector is a single element the NAMED_POINT translation is equivalent to a double translation.

- `BOOL -> (vectorNorm(val)!=0)?"1":"0"`

Conversion from Complex Vector

- `INT -> trunc(vectorNorm(val))`
- `DOUBLE -> vectorNorm(val)`
- `COMPLEX -> val[0]`
- `VECTOR -> [abs(val[0]),abs(val[1]),...abs(val[N])]?`
- `COMPLEX_VECTOR -> val`
- `NAMED_POINT -> {vectorString(val), NaN}`
- `STRING -> complexVectorString(val)`
- `BOOL -> (vectorNorm(val)!=0)?"1":"0"`

See [Conversion from vector double](#) for definitions of `vectorNorm`.

Conversion from Complex

- `INT -> trunc((val.imag() == 0)?val.real(): std::abs(val))`
- `DOUBLE -> val.imag() == 0)?val.real(): std::abs(val)`
- `COMPLEX -> val`
- `VECTOR -> [val.real,val.imag]`
- `COMPLEX_VECTOR -> [val]`
- `NAMED_POINT -> {helicsComplexString(val), NaN}`
- `STRING -> helicsComplexString(val)`
- `BOOL -> (std::abs(val)!=0)?"1":"0"`

If the imaginary value == 0, the value is treated the same as a double.

Conversion from a Named Point

If the value of the named point is NaN then treat the name part the same as a string, otherwise use the numerical value as a double and convert appropriately. The exception is a string which has a dedicated operation to generate a JSON string with two fields {"name" and "value"}.

Conversion from Bool

- `INT -> (val)?1:0`
- `DOUBLE -> (val)?1.0:0.0`
- `COMPLEX -> (val)?1.0:0.0 + 0.0j`
- `VECTOR -> [(val)?1.0:0.0]`
- `COMPLEX_VECTOR -> [(val)?1.0:0.0 +0.0j]`
- `NAMED_POINT -> {"value",(val)?1.0:0.0}`

- `STRING -> val?"1":"0";`
- `BOOL -> val?"1":"0";`

Conversion from Time

Time is transmitted as a 64 bit integer so conversion rules of an integer apply with the note that a double as a time is assumed as seconds and the integer represents nanoseconds so a double (as long as not too big) will be transmitted without loss as long as the precision is 9 decimal digits or less.

2.11.5 Unit conversions

HELICS also handles unit conversions if units are specified on the publication and subscription and can be understood by the units library. This applies primarily for pub/sub of numerical types. HELICS uses [Units](#) as the units library.

REFERENCES

3.1 Tools with HELICS Support

The following list of tools is a list of tools that have worked with HELICS at some level either on current projects or in the past, or in some cases funded projects that will be working with certain tools. These tools are in various levels of development. Check the corresponding links for more information.

3.1.1 Power Systems Tools

Electric Distribution System Simulation

- **GridLAB-D**, an open-source tool for distribution power-flow, DER models, basic house thermal and end-use load models, and more. HELICS support currently (8/15/2018) provided in the [develop branch](#) which you have to build yourself as described [here](#). Or a CMake based [branch](#) maintained as part of the [GMLC-TDC organization](#).
- **OpenDSS**, an open-source tool for distribution powerflow, DER models, harmonics, and other capabilities traditionally found in commercial distribution analysis tools. There are two primary interfaces with HELICS support:
 - [OpenDSSDirect.py](#) which provides a “direct” interface to interact with the OpenDSS engine enabling support for non-Windows (Linux, OSX) systems.
 - [PyDSS](#) which builds on OpenDSSDirect to provide enhanced advanced inverter models and significantly more robust convergence with high-penetration DER controls along with flexible support for user-defined controls and visualization.
- **CYME** has been used in connection with a python wrapper interface and through FMI wrapper.

Electric Transmission System Simulation

- **GridDyn**, an open-source transmission power flow and dynamics simulator. HELICS support provided through the [cmake_updates branch](#).
- **PSST**, an open-source python-based unit-commitment and dispatch market simulator. HELICS examples are included in the [HELICS-Tutorial](#).
- **MATPOWER**, an open-source Matlab based power flow and optimal power flow tool. HELICS support under development.
- **InterPSS**, a Java-based power systems simulator. HELICS support under development. [Use case instructions can be found here](#).
- **PSLF** has some level of support using the experimental python interface.
- **PSS/E**

- **PowerWorld** Simulator is an interactive power system simulation package designed to simulate high voltage power system operation on a time frame ranging from several minutes to several days.
- **PyPower** does not have a standard HELICS integration but it has been used on various projects. PYPOWER is a power flow and Optimal Power Flow (OPF) solver. It is a port of MATPOWER to the Python programming language. Current features include:
 - DC and AC (Newton’s method & Fast Decoupled) power flow and
 - DC and AC optimal power flow (OPF)

Real time simulators

- **OpalRT** A few projects are using HELICS to allow connections between Opal RT and other simulations
- **RTDS** Some planning or testing for RTDS linkages to HELICS is underway and will be required for some known projects

Electric Power Market simulation

- **FESTIV**, the Flexible Energy Scheduling Tool for Integrating Variable Generation, provides multi-timescale steady-state power system operations simulations that aims to replicate the full time spectrum of scheduling and reserve processes (multi-step commitment and dispatch plus simplified AGC) to meet energy and reliability needs of the bulk power system.
- **PLEXOS**, a commercial production cost simulator. Support via OpenPLEXOS is under development.
- **MATPOWER** (described above) also includes basic optimal powerflow support.
- **PyPower** (described above) also includes basic AC and DC optimal powerflow solvers.

Contingency Analysis tools

- **CAPE** protection system modeling.
- **DCAT** Dynamic contingency analysis tool.

3.1.2 Communication Tools

- HELICS provides built-in support for simple communications manipulations such as delays, lossy channels, etc. through its built-in filters.
- **ns-3**, a discrete-event communication network simulator. Supported via the **HELICS ns-3 module**.
- **OMNet++** is a public-source, component-based, modular and open-architecture simulation environment with strong GUI support and an embeddable simulation kernel. Its primary application area is the simulation of communication networks, but it has been successfully used in other areas like the simulation of IT systems, queueing networks, hardware architectures and business processes as well. Early stage development with OMNET++ and HELICS is underway and a prototype example is available in **HELICS-omnetpp**.

3.1.3 Gas Pipeline Modeling

- NGFAST.
- GasModels.jl.

3.1.4 Optimization packages

- GAMS.
- JuMP support is provided through the HELICS Julia interface.

3.1.5 Transportation modeling

- BEAM.
- POLARIS.

3.2 Built-In HELICS Apps

Included with HELICS are a number of apps that provide useful utilities and test programs for getting started and running with HELICS

3.2.1 Recorder

The Recorder application is one of the HELICS apps available with the library Its purpose is to provide a easy way to capture data from a federation It acts as a federate that can “capture” values or messages from specific publications or direct endpoints or cloned endpoints which exist elsewhere

Command line arguments

allowed options:

command line only:

-? [--help]	produce help message
-v [--version]	display a version string
--config-file arg	specify a configuration file to use

configuration:

--local	specify otherwise unspecified endpoints and publications as local(i.e.the keys will be prepended with the player name
--stop arg	the time to stop the player
--quiet	turn off most display output

allowed options:

configuration:

-b [--broker] arg	address of the broker to connect
---------------------	----------------------------------

(continues on next page)

(continued from previous page)

```

-n [ --name ] arg      name of the player federate
--corename arg         the name of the core to create or find
-c [ --core ] arg      type of the core to connect to
--offset arg           the offset of the time steps
--period arg           the period of the federate
--timedelta arg        the time delta of the federate
--rttolerance arg      the time tolerance of the real time mode
-i [ --coreinit ] arg  the core initialization string
--separator arg        separator character for local federates
--inputdelay arg       the input delay on incoming communication of the
                        federate
--outputdelay arg      the output delay for outgoing communication of the
                        federate
-f [ --flags ] arg     named flag for the federate

```

allowed options:

configuration:

```

--tags arg             tags to record, this argument may be specified any
                        number of times
--endpoints arg        endpoints to capture, this argument may be specified
                        multiple time
--sourceclone arg      existing endpoints to capture generated packets from,
                        this argument may be specified multiple time
--destclone arg        existing endpoints to capture all packets with the
                        specified endpoint as a destination, this argument may
                        be specified multiple time
--clone arg            existing endpoints to clone all packets to and from
--capture arg          capture all the publications of a particular federate
                        capture="fed1;fed2" supports multiple arguments or a
                        semicolon/comma separated list
-o [ --output ] arg    the output file for recording the data
--allow_iteration       allow iteration on values
--verbose              print all value results to the screen
--marker arg           print a statement indicating time advancement      every <arg>
↪seconds of the simulation
                        is the period of the marker
--mapfile arg          write progress to a map file for concurrent progress
                        monitoring

```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the player executable also takes an untagged argument of a file name for example

```
helics_recorder record_file.txt --stop 5
```

Recorders support both delimited text files and json files some examples can be found in

Player configuration examples

config file detail

subscriptions

a simple example of a recorder file specifying some subscriptions

```
#FederateName topic1

sub pub1
subscription pub2
```

signifies a comment

if only a single column is specified it is assumed to be a subscription

for two column rows the second is the identifier arguments with spaces should be enclosed in quotes

interface	description
s, sub, subscription	subscribe to a particular publication
endpoint, ept, e	generate an endpoint to capture all targeted packets
source, sourceclone,src	capture all messages coming from a particular endpoint
dest, destination, destclone	capture all message going to a particular endpoint
capture	capture all data coming from a particular federate
clone	capture all message going from or to a particular endpoint

for 3 column rows the first must be either clone or capture for clone the second can be either source or destination and the third the endpoint name [for capture it can be either “endpoints” or “subscriptions”] NOTE: not fully working yet for capture

JSON configuration

recorders can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#list publications and endpoints for a recorder

pub1
pub2
e src1
```

JSON example

```
{
  "subscriptions": [
    {
      "key": "pub1",
      "type": "double"
    },
    {
      "key": "pub2",
      "type": "double"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

],
"endpoints": [
  {
    "name": "src1",
    "global": true
  }
]
}

```

some configuration can also be done through JSON through elements of “stop”, “local”, “separator”, “timeunits” and file elements can be used to load up additional files

output

Recorders capture files in a format the Player can read see *Player* the `--verbose` option will also print the values to the screen.

Map file output

the recorder can generate a live file that can be used in process to see the progress of the Federation This is occasionally useful, though for many uses the *Tracer* will be more useful when it is completed

3.2.2 Player

The player application is one of the HELICS apps available with the library Its purpose is to provide a easy way to generate data into a federation It acts as a federate that can “play” values or messages at specific times It exists as a standalone executable but also as library object so could be integrated into other components

Command line arguments

```

command line only:
  -? [ --help ]           produce help message
  -v [ --version ]        display a version string
  --config-file arg       specify a configuration file to use

configuration:
  --local                  specify otherwise unspecified endpoints and
                           publications as local( i.e.the keys will be prepended
                           with the player name
  --stop arg               the time to stop the player
  --quiet                  turn off most display output

configuration:
  -b [ --broker ] arg     address of the broker to connect
  -n [ --name ] arg       name of the player federate
  --corename arg           the name of the core to create or find
  -c [ --core ] arg       type of the core to connect to

```

(continues on next page)

(continued from previous page)

<code>--offset arg</code>	the offset of the time steps
<code>--period arg</code>	the period of the federate
<code>--timedelta arg</code>	the time delta of the federate
<code>--rttolerance arg</code>	the time tolerance of the real time mode
<code>-i [--coreinit] arg</code>	the core initialization string
<code>--separator arg</code>	separator character for local federates
<code>--inputdelay arg</code>	the input delay on incoming communication of the federate
<code>--outputdelay arg</code>	the output delay for outgoing communication of the federate
<code>-f [--flags] arg</code>	named flag for the federate
allowed options:	
configuration:	
<code>--datatype arg</code>	type of the publication data type to use
<code>--marker arg</code>	print a statement indicating time advancement every <code>arg</code> seconds is the period of the marker
<code>--time_units arg</code>	the default units on the timestamps used in file based input

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the player executable also takes an untagged argument of a file name for example

```
helics_player player_file.txt --stop 5
```

Players support both delimited text files and JSON files some examples can be found in

Player configuration examples

Config File Detail

publications

a simple example of a player file publishing values

#second	topic	type(opt)	value
-1.0,	pub1, d,	0.3	
1,	pub1,	0.5	
3,	pub1	0.8	
2,	pub1	0.7	
# pub 2			
1,	pub2, d,	0.4	
2,	pub2,	0.6	
3,	pub2,	0.9	
4,	0.7	# this statement is assumed to refer to pub 2	

signifies a comment the first column is time in seconds unless otherwise specified via the `--time_units` flag or other configuration means the second column is publication name the final column is the value the optional third column specifies a type valid types are

time specifications are typically numerical with optional units 5 or "500 ms" or 23.7us if there is a space between the number and units it must be enclosed in quotes if no units are specified the time defaults to units specified via --time_units or seconds if none were specified valid units are "s", "ms", "us", "min", "day", "hr", "ns", "ps" the default precision in HELICS is ns so time specified in ps is not guaranteed to be precise

identifier	type	Example
d,f, double	double	45.1
s,string	string	"this is a test"
i, i64, int	integer	456
c, complex	complex	23+2j, -23.1j, 1+3i
v, vector	vector of doubles	[23.1,34,17.2,-5.6]
cv, complex_vector	vector of complex numbers	[23+2j, -23.1j, 1+3i]

capitalization does not matter

values with times <0 are sent during the initialization phase values with time==0 are sent immediately after entering execution phase

Messages

messages are specified in one of two forms

```
m <time> <source> <dest> <data>
```

or

```
m <sendtime> <deliverytime> <source> <dest> <time> <data>
```

the second option allows sending events at a different time than they are triggered the data portion of messages can be encoded in base64 by marking as b64[] or base64[X] all data between the brackets will be converted to raw binary. A ']' must be last. The string interpreter can also handle messages with any escapable characters including tab ("t"), newline ("n"), and quote ("\""), this can be marked by using quotes as in "<message>" to make it interpret the message as a JSON quoted string.

JSON configuration

player values can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#example player file
mess 1.0 src dest "this is a test message"
mess 1.0 2.0 src dest "this is test message2"
M 2.0 3.0 src dest "this is message 3"
```

JSON example

```
{
  "messages": [{
    "source": "src",
    "dest": "dest",
    "time": 1.0,
```

(continues on next page)

(continued from previous page)

```

        "data": "this is a test message"
    }, {
        "source": "src",
        "dest": "dest",
        "time": 1.0,
        "encoding": "base64"
        "data":
        ↪AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8gISIjJCUmJygpKissLS4vMDEyMzQ1Njc4OT07PD0+P0BBQkNERUZHSElKS
        ↪wMHCw8TFxsfIycrLzM30z9DR0tPU1dbX2Nna29zd3t/g4eLj50Xm5+jp6uvs7e7v8PHy8/T19vf4+fr7/P3+/
        ↪w=="
    }, {
        "source": "src",
        "dest": "dest",
        "time": 2.0,
        "data": "this is test message2"
    }, {
        "source": "src",
        "dest": "dest",
        "time": 3.0,
        "data": "this is message 3"
    }
}

```

#second	topic	type(opt)	value
-1	pub1	d	0.3
1	pub1	d	0.5
2	pub1	d	0.7
3	pub1	d	0.8
1	pub2	d	0.4
2	pub2	d	0.6
3	pub2	d	0.9

Example JSON

```

{
  "points": [
    {
      "key": "pub1",
      "type": "double",
      "value": 0.3,
      "time": -1
    },
    {
      "key": "pub2",
      "type": "double",
      "value": 0.4,
      "time": 1.0
    },
    {
      "key": "pub1",
      "value": 0.5,

```

(continues on next page)

(continued from previous page)

```

    "time": 1.0
  },
  {
    "key": "pub1",
    "value": 0.8,
    "time": 3.0
  },
  {
    "key": "pub1",
    "value": 0.7,
    "time": 2.0
  },
  {
    "key": "pub2",
    "value": 0.6,
    "time": 2.0
  },
  {
    "key": "pub2",
    "value": 0.9,
    "time": 3.0
  }
]
}

```

some configuration can also be done through JSON through elements of “stop”, “local”, “separator”, “time_units” and file elements can be used to load up additional files

3.2.3 Source

The Source app generates signals for other federates, it functions similarly to the player but doesn’t take a prescribed file instead it generates signals according to some mathematical function, like sine, ramp, pulse, or random walk. This can be useful for sending probing signals or just testing responses of the federate to various stimuli.

Command line arguments

allowed options:

command line only:

-? [--help]	produce help message
-v [--version]	display a version string
--config-file arg	specify a configuration file to use

configuration:

--datatype arg	type of the publication data type to use
--local	specify otherwise unspecified endpoints and publications as local(i.e.the keys will be prepended with the player name
--separator arg	specify the separator for local publications and endpoints

(continues on next page)

(continued from previous page)

<code>--time_units arg</code>	the default units on the timestamps used in file based input
<code>--stop arg</code>	the time to stop the player
federate configuration	
<code>-b [--broker] arg</code>	address of the broker to connect
<code>-n [--name] arg</code>	name of the player federate
<code>--corename arg</code>	the name of the core to create or find
<code>-c [--core] arg</code>	type of the core to connect to
<code>--offset arg</code>	the offset of the time steps
<code>--period arg</code>	the period of the federate
<code>--timedelta arg</code>	the time delta of the federate
<code>-i [--coreinit] arg</code>	the core initialization string
<code>--inputdelay arg</code>	the input delay on incoming communication of the federate
<code>--outputdelay arg</code>	the output delay for outgoing communication of the federate
<code>-f [--flags] arg</code>	named flags for the federate

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the player executable also takes an untagged argument of a file name for example

```
helics_player player_file.txt --stop 5
```

Players support both delimited text files and JSON files some examples can be found in

Player configuration examples

Config File Detail

publications

a simple example of a player file publishing values

#second	topic	type(opt)	value
-1.0,	pub1, d,	0.3	
1,	pub1,	0.5	
3,	pub1	0.8	
2,	pub1	0.7	
# pub 2			
1,	pub2, d,	0.4	
2,	pub2,	0.6	
3,	pub2,	0.9	

signifies a comment the first column is time in seconds unless otherwise specified via the `--time_units` flag or other configuration means the second column is publication name the final column is the value the optional third column specifies a type valid types are

time specifications are typically numerical with optional units 5 or "500 ms" or 23.7us if there is a space between the number and units it must be enclosed in quotes if no units are specified the time defaults to units specified via

--time_units or seconds if none were specified valid units are “s”, “ms”, “us”, “min”, “day”, “hr”, “ns”, “ps” the default precision in HELICS is ns so time specified in ps is not guaranteed to be precise

identifier	type	Example
d,f, double	double	45.1
s,string	string	“this is a test”
i, i64, int	integer	456
c, complex	complex	23+2j, -23.1j, 1+3i
v, vector	vector of doubles	[23.1,34,17.2,-5.6]
cv, complex_vector	vector of complex numbers	[23+2j, -23.1j, 1+3i]

capitalization does not matter

values with times <0 are sent during the initialization phase values with time==0 are sent immediately after entering execution phase

Messages

messages are specified in one of two forms

```
m <time> <source> <dest> <data>
```

or

```
m <sendtime> <deliverytime> <source> <dest> <time> <data>
```

the second option allows sending events at a different time than they are triggered

JSON configuration

player values can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#example player file
mess 1.0 src dest "this is a test message"
mess 1.0 2.0 src dest "this is test message2"
M 2.0 3.0 src dest "this is message 3"
```

JSON example

```
{
  "messages": [
    {
      "source": "src",
      "dest": "dest",
      "time": 1.0,
      "data": "this is a test message"
    },
    {
      "source": "src",
      "dest": "dest",
```

(continues on next page)

(continued from previous page)

```

    "time": 2.0,
    "data": "this is test message2"
  },
  {
    "source": "src",
    "dest": "dest",
    "time": 3.0,
    "data": "this is message 3"
  }
]
}

```

#second	topic	type(opt)	value
-1	pub1 d	0.3	
1	pub1 d	0.5	
2	pub1 d	0.7	
3	pub1 d	0.8	
1	pub2 d	0.4	
2	pub2 d	0.6	
3	pub2 d	0.9	

Example JSON

```

{
  "points": [
    {
      "key": "pub1",
      "type": "double",
      "value": 0.3,
      "time": -1
    },
    {
      "key": "pub2",
      "type": "double",
      "value": 0.4,
      "time": 1.0
    },
    {
      "key": "pub1",
      "value": 0.5,
      "time": 1.0
    },
    {
      "key": "pub1",
      "value": 0.8,
      "time": 3.0
    },
    {
      "key": "pub1",
      "value": 0.7,
      "time": 2.0
    }
  ],

```

(continues on next page)

(continued from previous page)

```
{
  {
    "key": "pub2",
    "value": 0.6,
    "time": 2.0
  },
  {
    "key": "pub2",
    "value": 0.9,
    "time": 3.0
  }
}
```

some configuration can also be done through JSON through elements of “stop”, “local”, “separator”, “time_units” and file elements can be used to load up additional files

3.2.4 helics_app

The HELICS apps executable is one of the HELICS apps available with the library Its purpose is to provide a common executable for running any of the other as

typical syntax is as follows

```
helics-app.exe <app> <app arguments ...>
```

possible apps are

Echo

The *Echo* app is a responsive app that will echo any message sent to its endpoints back to the original source with a specified delay

This is useful for testing communication pathways and in combination with filters can be used to create some interesting situations

Player

The *Player* app will generate signals through specified interfaces from prescribed data This is used for generating test signals into a federate

Recorder

The *Recorder* app captures signals and data on specified interfaces and can record then to various file formats including text files and JSON files The files saved can then be used by the Player app at a later time

Tracer

The *Tracer* app functions much like the recorder when run as a standalone app with the exception that it displays information to a text window and doesn't capture to a file. The additional purpose is used as a library object as the basis for additional display purposes and interfaces.

Source

The *Source* app is a signal generator like the player except that it can generate signals from defined patterns including some random signals in value and timing, and other patterns like sine, square wave, ramps and others. Used much like the player in situations some test signals are needed.

Broker

The *Broker* executes a broker like the stand alone Broker app, it does not include the broker terminal application.

Clone

The *Clone* has the ability to copy another federate and record it to a file that can be used by a Player. It will duplicate all publications and subscriptions of a federate.

MultiBroker

The Multibroker is an in progress development of a broker that can interact with multiple communication modes. Such as a single broker that can act as a bridge between MPI and ZeroMQ or other network protocols. More documentation will be available as the multibroker is developed.

3.2.5 Command Line Arguments

allowed options:	
command line only:	
-? [--help]	produce help message
-v [--version]	display a version string
--config-file arg	specify a configuration file to use
configuration:	
-n [--name] arg	name of the broker
-t [--type] arg	type of the broker ("zmq", "ipc", "test", "mpi", "test", "tcp", "udp")
Help for Zero MQ Broker:	
configuration:	
--interface arg	the local interface to use for the receive ports
-b [--broker] arg	identifier for the broker
--broker_address arg	location of the broker i.e network address
--brokerport arg	port number for the broker priority port
--localport arg	port number for the local receive port

(continues on next page)

(continued from previous page)

```
--port arg          port number for the broker's port
--portstart arg     starting port for automatic port definitions
```

Help for Interprocess Broker:

configuration:

```
--queueloc arg      the named location of the shared queue
-b [ --broker ] arg  identifier for the broker
--broker_address arg location of the broker i.e network address
--brokerinit arg     the initialization string for the broker
```

Help for Test Broker:

configuration:

```
--brokername arg    identifier for the broker-same as broker
-b [ --broker ] arg  identifier for the broker
--broker_address arg location of the broker i.e network address
--brokerinit arg     the initialization string for the broker
```

Help for UDP Broker:

configuration:

```
--interface arg     the local interface to use for the receive ports
-b [ --broker ] arg  identifier for the broker
--broker_address arg location of the broker i.e network address
--brokerport arg     port number for the broker priority port
--localport arg      port number for the local receive port
--port arg           port number for the broker's port
--portstart arg      starting port for automatic port definitions
```

3.2.6 Echo

The Echo application is one of the HELICS apps available with the library. Its purpose is to provide a easy way to generate an echo response to a message. Mainly for testing and demos.

Command line arguments

allowed options:

command line only:

```
-? [ --help ]      produce help message
-v [ --version ]    display a version string
--config-file arg   specify a configuration file to use
```

configuration:

```
--local            specify otherwise unspecified endpoints and
                   publications as local( i.e.the keys will be prepended
                   with the echo name
--stop arg         the time to stop the app
```

(continues on next page)

(continued from previous page)

```

configuration:
  -b [ --broker ] arg      address of the broker to connect
  -n [ --name ] arg        name of the player federate
  --corename arg           the name of the core to create or find
  -c [ --core ] arg        type of the core to connect to
  --offset arg             the offset of the time steps
  --period arg             the period of the federate
  --timedelta arg          the time delta of the federate
  -i [ --coreinit ] arg    the core initialization string
  --separator arg          separator character for local federates
  --inputdelay arg         the input delay on incoming communication of the
                           federate
  --outputdelay arg        the output delay for outgoing communication of the
                           federate
  -f [ --flags ] arg       named flag for the federate

configuration:
  --delay arg              the delay with which the echo app will echo message

```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the echo executable also takes an untagged argument of a file name for example

```
helics_app echo echo_file.txt --stop 5
```

The Echo app supports JSON files some examples can be found in

Echo configuration examples

the main property of the echo app is the delay time which messages are echoed.

3.2.7 Tracer

The Tracer application is one of the HELICS apps available with the library Its purpose is to provide a easy way to display data from a federation It acts as a federate that can “capture” values or messages from specific publications or direct endpoints or cloned endpoints which exist elsewhere and either trigger callbacks or display it to a screen The main use is a simple visual indicator and a monitoring app

Command line arguments

```

allowed options:

command line only:
  -? [ --help ]           produce help message
  -v [ --version ]        display a version string
  --config-file arg       specify a configuration file to use

configuration:

```

(continues on next page)

(continued from previous page)

<code>--stop arg</code>	the time to stop recording
<code>--tags arg</code>	tags to record, this argument may be specified any number of times
<code>--endpoints arg</code>	endpoints to capture, this argument may be specified multiple time
<code>--sourceclone arg</code>	existing endpoints to capture generated packets from, this argument may be specified multiple time
<code>--destclone arg</code>	existing endpoints to capture all packets with the specified endpoint as a destination, this argument may be specified multiple time
<code>--clone arg</code>	existing endpoints to clone all packets to and from
<code>--capture arg</code>	capture all the publications of a particular federate <code>capture="fed1;fed2"</code> supports multiple arguments or a semicolon/comma separated list
<code>-o [--output] arg</code>	the output file for recording the data
<code>--mapfile arg</code>	write progress to a memory mapped file
federate configuration	
<code>-b [--broker] arg</code>	address of the broker to connect
<code>-n [--name] arg</code>	name of the player federate
<code>--corename arg</code>	the name of the core to create or find
<code>-c [--core] arg</code>	type of the core to connect to
<code>--offset arg</code>	the offset of the time steps
<code>--period arg</code>	the period of the federate
<code>--timedelta arg</code>	the time delta of the federate
<code>-i [--coreinit] arg</code>	the core initialization string
<code>--inputdelay arg</code>	the input delay on incoming communication of the federate
<code>--outputdelay arg</code>	the output delay for outgoing communication of the federate
<code>-f [--flags] arg</code>	named flags for the federate

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the tracer executable also takes an untagged argument of a file name for example

```
helics_app tracer tracer_file.txt --stop 5
```

Tracers support both delimited text files and JSON files some examples can be found in, they are otherwise the same as options for recorders.

Tracer configuration examples

Config File Detail

subscriptions

a simple example of a recorder file specifying some subscriptions

```
#FederateName topic1

sub pub1
subscription pub2
```

signifies a comment

if only a single column is specified it is assumed to be a subscription

for two column rows the second is the identifier arguments with spaces should be enclosed in quotes

interface	description
s, sub, subscription	subscribe to a particular publication
endpoint, ept, e	generate an endpoint to capture all targeted packets
source, sourceclone,src	capture all messages coming from a particular endpoint
dest, destination, destclone	capture all message going to a particular endpoint
capture	capture all data coming from a particular federate
clone	capture all message going from or to a particular endpoint

for 3 column rows the first must be either clone or capture for clone the second can be either source or destination and the third the endpoint name [for capture it can be either “endpoints” or “subscriptions”]

JSON configuration

Tracers can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#list publications and endpoints for a recorder

pub1
pub2
e src1
```

JSON example

```
{
  "subscriptions": [
    {
      "key": "pub1",
      "type": "double"
    },
    {
      "key": "pub2",
      "type": "double"
    }
  ],
}
```

(continues on next page)

(continued from previous page)

```

"endpoints": [
  {
    "name": "src1",
    "global": true
  }
]
}

```

some configuration can also be done through JSON through elements of “stop”, “local”, “separator”, “timeunits” and file elements can be used to load up additional files

3.2.8 Broker

Brokers function as intermediaries or roots in the HELICS hierarchy. The Broker can be run through the `helics_broker` or via `helics-app`

Command line arguments

```

helics_broker term <broker args...> will start a broker and open a terminal control_
↳ window for the broker run help in a terminal for more commands
helics_broker --autorestart <broker args ...> will start a continually regenerating_
↳ broker there is a 3 second countdown on broker completion to halt the program via ctrl-
↳ C
helics_broker <broker args ...> just starts a broker with the given args and waits for it_
↳ to complete
allowed options:

command line only:
-? [ --help ]           produce help message
-v [ --version ]        display a version string
--config-file arg       specify a configuration file to use

configuration:
-n [ --name ] arg       name of the broker
-t [ --type ] arg       type of the broker ("zmq", "ipc", "test", "mpi", "test", "tcp",
↳ "udp")

Help for Zero MQ Broker:
allowed options:

configuration:
--interface arg         the local interface to use for the receive ports
-b [ --broker ] arg     identifier for the broker
--broker_address arg    location of the broker i.e network address
--brokername arg        the name of the broker
--local                 use local interface(default)
--ipv4                  use external ipv4 addresses
--ipv6                  use external ipv6 addresses
--external              use all external interfaces
--brokerport arg        port number for the broker priority port

```

(continues on next page)

(continued from previous page)

```
--localport arg      port number for the local receive port
--port arg           port number for the broker's port
--portstart arg      starting port for automatic port definitions
```

Help for Interprocess Broker:

allowed options:

configuration:

```
--queueloc arg      the named location of the shared queue
-b [ --broker ] arg  identifier for the broker
--broker_address arg location of the broker i.e network address
--brokerinit arg     the initialization string for the broker
```

Help for Test Broker:

allowed options:

configuration:

```
--brokername arg     identifier for the broker-same as broker
-b [ --broker ] arg  identifier for the broker
--broker_address arg  location of the broker i.e network address
--brokerinit arg     the initialization string for the broker
```

Help for TCP Broker:

allowed options:

configuration:

```
--interface arg      the local interface to use for the receive ports
-b [ --broker ] arg  identifier for the broker
--broker_address arg  location of the broker i.e network address
--brokername arg     the name of the broker
--local              use local interface(default)
--ipv4               use external ipv4 addresses
--ipv6               use external ipv6 addresses
--external           use all external interfaces
--brokerport arg     port number for the broker priority port
--localport arg      port number for the local receive port
--port arg           port number for the broker's port
--portstart arg      starting port for automatic port definitions
```

Help for UDP Broker:

allowed options:

configuration:

```
--interface arg      the local interface to use for the receive ports
-b [ --broker ] arg  identifier for the broker
--broker_address arg  location of the broker i.e network address
--brokername arg     the name of the broker
--local              use local interface(default)
--ipv4               use external ipv4 addresses
--ipv6               use external ipv6 addresses
--external           use all external interfaces
--brokerport arg     port number for the broker priority port
```

(continues on next page)

(continued from previous page)

<code>--localport arg</code>	port number for the local receive port
<code>--port arg</code>	port number for the broker's port
<code>--portstart arg</code>	starting port for automatic port definitions
Broker Specific options:	
configuration:	
<code>--root</code>	specify whether the broker is a root
configuration:	
<code>-n [--name] arg</code>	name of the broker/core
<code>--federates arg</code>	the minimum number of federates that will be connecting
<code>--minfed arg</code>	the minimum number of federates that will be connecting
<code>--maxiter arg</code>	maximum number of iterations
<code>--logfile arg</code>	the file to log message to
<code>--loglevel arg</code>	the level which to log the higher this is set to the more gets logs (-1) for no logging
<code>--fileloglevel arg</code>	the level at which messages get sent to the file
<code>--consoleloglevel arg</code>	the level at which message get sent to the console
<code>--minbrokers arg</code>	the minimum number of core/brokers that need to be connected (ignored in cores)
<code>--identifier arg</code>	name of the core/broker
<code>--tick arg</code>	number of milliseconds per tick counter if there is no broker communication for 2 ticks then secondary actions are taken (can also be entered as a time like '10s' or '45ms')
<code>--dumplog</code>	capture a record of all messages and dump a complete log to
<code>↪ file or console on termination</code>	
<code>--terminate_on_error</code>	Specify that the co-simulation should terminate if any error
<code>↪ occurs</code>	
<code>--timeout arg</code>	milliseconds to wait for a broker connection (can also be entered as a time like '10s' or '45ms')
<code>--error_timeout arg</code>	milliseconds to wait before disconnecting after an error (can also be entered as a time like '10s' or '45ms')

If the Broker is started with `term` as the first option, a terminal is opened for user entry of commands all command line arguments following `term` are passed to the broker.

```
starting broker
helics>>help
`quit` -> close the terminal application and wait for broker to finish
`terminate` -> force the broker to stop
`terminate*` -> force the broker to stop and exit application
`help`,`?` -> this help display
`restart` -> restart a completed broker
`status` -> will display the current status of the broker
`info` -> will display info about the broker
`force restart` -> will force terminate a broker and restart it
`query` <queryString> -> will query a broker for <queryString>
`query` <queryTarget> <queryString> -> will query <queryTarget> for <queryString>
```

(continues on next page)

(continued from previous page)

```
helics>>
```

status will print out current status of the brokers including counts of federates, brokers, and handles

```
helics>>status
Broker (643204-ibrVd-14EWH-unKfh-hExUP) is connected and is accepting new federates
{"brokers":0,
"federates":0,
"handles":0}
helics>>
```

info prints out name, connection status, and connection information

```
helics>>info
Broker (643204-ibrVd-14EWH-unKfh-hExUP) is connected and is accepting new federates
address=tcp://127.0.0.1:23404
```

The query command allows any query to be executed from the command line, query counts displays the same count numbers as status.

Other available queries are described in *Queries*.

various restart options are also available, terminate, restart, force restart. And finally quit will exit the terminal and wait for the broker to complete. enter terminate before quit or terminate* to terminate and quit.

3.2.9 Broker Server

Brokers function as intermediaries or roots in the HELICS hierarchy The broker server is an executable that can be used to automatically generate brokers on an as needed basis and coordinate their control and management. It is considered experimental as version 2.2 only works with the ZMQ core type. Future versions will expand this significantly.

Future plans include expanding to all networking core types (ZMQ, ZMQSS, TCP, TCPSS, UDP, and MPI), expanding the abilities of a terminal program and making a Restful interface to the server and underlying brokers.

Command line arguments

The Broker server is a helics broker coordinator that can generate brokers on request
Usage:helics_broker_server [OPTIONS] [config]

Positionals:

config TEXT	load a config file for the broker server
-------------	--

Options:

-h,-?,--help	Print this help message and exit
-v,--version	
-z,--zmq	start a broker-server for the zmq comms in helics
--zmqss	start a broker-server for the zmq single socket comms in
helics	
-t,--tcp	start a broker-server for the tcp comms in helics
-u,--udp	start a broker-server for the udp comms in helics
--mpi	start a broker-server for the mpi comms in helics

[Option Group: quiet]

(continues on next page)

(continued from previous page)

```

Options:
  --quiet                      silence most print output

helics broker server command line
helics_broker_server [OPTIONS] [SUBCOMMAND]

Options:
  -h,-?,--help                Print this help message and exit
  -v,--version
  --duration TIME=30 minutes  specify the length of time the server should run
[Option Group: quiet]
Options:
  --quiet                      silence most print output

Subcommands:
  term                        helics_broker_server term will start a broker server and
  ↪ open a terminal control window for the broker server, run help in a terminal for more
  ↪ commands

helics_broker_server server types starts a broker with the given args and waits for it
  ↪ to complete

```

If the Broker_server is started with `term` as the first option, a terminal is opened for user entry of commands all command line arguments following `term` are passed to the broker.

```

starting broker Server
servers started
helics-broker-server>>help
`quit` -> close the terminal application and wait for broker to finish
`terminate` -> force the broker server to stop
`terminate*` -> force the broker server to stop and all existing brokers to terminate
`help`,`?` -> this help display

helics-broker-server>>

```

more commands will be added in future releases

3.2.10 Clone

The Clone application is one of the HELICS apps available with the library Its purpose is to provide a easy way to clone a federate for later playback It acts as a federate that can “capture” values or messages from a single federate It also captures the interfaces and subscriptions of a federate and will store those in a configuration file that can be used by the *Player*. The clone app will try to match the federate being cloned as close as possible in timing of messages and publications and subscriptions. At present it does not match nameless publications or filters.

Command line arguments

```
Helics Clone App
Usage: helics_app clone [OPTIONS]

Command line options for the Clone App
Usage: [OPTIONS] [capture]

Positionals:
  capture TEXT          name of the federate to clone

Options:
  --allow_iteration      allow iteration on values
  -o,--output TEXT=clone.json the output file for recording the data

Options:
  -h,-?,--help          Print this help message and exit
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the clone app is accessible through the helics_app

```
helics_app clone fed1 -o fed1.json -stop 10
```

output

The Clone app captures output and configuration in a JSON format the *Player* can read. All publications of a federate are created as global with the name of the original federate, so a player could be named something else if desired and not impact the transmission.

3.3 Configuration Options Reference

Many of the HELICS entities have significant configuration options. Rather than comprehensively list these options while explaining the features themselves, we've created this section of the User Guide to serve as a reference as to what they are, what they do, and how to use them. This reference is oriented-around the use of JSON files for configuration and is an attempt to be comprehensive in listing and explaining those options. As will be explained below, many of these options are accessible via direct API calls though some of these calls are general in nature (such as helicsFederateInfoSetIntegerProperty to set the logging level, among other things). As such

3.3.1 Configuration methods

Generally, there are three ways in which a co-simulation can be configured and all the various options can be defined:

1. Using direct API calls in the federate source code.
2. Using command-line switches/flags which beginning execution of the federate
3. Using a JSON configuration file (and calling helicsCreateValueFederateFromConfig, helicsCreateMessageFederateFromConfig, or helicsCreateComboFederateFromConfig)

Not all configuration options are available in all three forms but often they are. For example, it is not possible (nor convenient) to configure a federate from the command line but it is possible to do so with the JSON config file and with API calls.

Choosing configuration method

Which method you use to configure your federate and co-simulation significantly depends on the circumstances of the co-simulation and details of any existing code-base being used. Here is some guidance, though, to help in guiding you're decision in how to do this:

- **If possible, use a JSON configuration file** - Using a JSON configuration file allows creates separation between the code base of the federation and its use in a particular co-simulation. This allows for a modularity between the functionality the federate provides and the particular co-simulation in which it is applied. For example, a power system federate can easily be reconfigured to work on one model vs another through the use of a JSON configuration file. The particular publications and subscriptions may change but the main functionality of the federate (solving the power flow) does not. To use the JSON file for configuration, one of three specific APIs needs to be called: in the file:
 - `helicsCreateValueFederateFromConfig` [C++](#) | [C](#) | [Python](#) | [Julia](#)
 - `helicsCreateMessageFederateFromConfig` [C++](#) | [C](#) | [Python](#) | [Julia](#)
 - `helicsCreateCombinationFederateFromConfig` [C++](#) | [C](#) | [Python](#) | [Julia](#)
- **JSON configuration produces a natural artifact that defines the co-simulation** - Another advantage of the external configuration in the JSON file is that it is a human-readable artifact that can be distributed separately from the source code that provides a lot of information about how the co-simulation was run. In fact, its possible to just look at the configuration files of a federation and do some high-level debugging (checking to see that the subscriptions and publications are aligned, for example).
- **New federates in ill-defined use cases may benefit from API configuration** - The modularity that the JSON config provides may not offer many benefits if the federate is newly integrated into HELICS and/or is part of an evolving analysis. During these times the person(s) doing the integration may just want to make direct API calls instead of having to mess with writing the federate code and a configuration file. There will likely be a point in the future when the software is more codified and switching to a JSON configuration makes more sense.
- **Command-line configuration (where possible) allows for small, quick changes to the configuration** - Because the command line doesn't provide comprehensive access to the necessary configuration, it will never be a stand-alone configuration option but it does have the advantage of providing quick access right as a user is instantiating the federate. This is an ideal place to make small changes to the configuration (e.g. changing the minimum time step) without having to edit any files.
- **API configuration is most useful for dynamic configuration** - If there is a need to change the configuration of a given federate dynamically, the API is the only way to do that. Such needs are not common but there are cases where, for example, it may be necessary to define the configuration based on the participants in the federation (e.g. publications, subscriptions, timing). It's possible to use *queries* to understand the composition and configuration of the federation and then use the APIs to define the configuration of the federate in question.

How to Use This Reference

The remainder of this reference lists the configuration options that are supported in the JSON configuration file. Where possible, the corresponding C++ API calls and the links to that documentation will be provided. Generally, the command-line options use the exact same syntax as the JSON configuration options preceded by a `--` and followed by either an `=` or a space and then the parameter value (*i.e.* `--name testname`). In the cases where a single letter switch is available, that will be listed (*i.e.* `-n testname`).

Default values are shown in “[]” following the name(s) of the option.

When an API exists, its name is shown along with links to the specific API documentation for a few (but, sadly, not all) of the supported languages. Many of the options are set with generic functions (*e.g.* `helicsFederateInfoSetFlagOption`) and in those cases the option being set is specified by an enumerated value. In C, these values (*e.g.* `helics_flag_uninterruptible`) are set to integer value (*e.g.* 1); in this document that integer value follows the enumeration string in brackets. If using the C interface, the integer value must be used. The C++ interface supports the use of the enumerated value directly as do the Python and Julia interfaces with slight syntactical variations (Python: `helics.HELICS_FLAG_INTERRUPTIBLE` and Julia: `HELICS.HELICS_FLAG_INTERRUPTIBLE`).

x

3.3.2 Sample Configurations

The JSON configuration file below shows all the configuration options in a single file along with their default values and shows what section of the file they should be placed in. Most JSON configuration files will require far fewer options than shown here; items marked with “***” are required. Many items have alternative names that are

Many of the configuration parameters have alternate names that provide the same functionality. Only one of the names is shown in this configuration file but the alternative names are listed in the reference below. Generally, the supported names are the same string in `nocase`, `camelCase`, and `snake_case`.

An example of one publication, subscription, named input, endpoint, and filter is also shown. The values for each of these options is arbitrary and in the case of filters, many more values are supported and a description of each is provided.

(Note that JSON does not support comments and thus the block below is not valid JSON.)

```
{
  // General
  ***"name": "arbitrary federate name",**
  "core_type": "zmq",
  "core_name": "core name",
  "core_init_string" : "",
  "autobroker": false,
  "connection_required": false,
  "connection_optional": false,
  "strict_input_type_checking": false,
  "terminate_on_error": false,
  "source_only": false,
  "observer": false,
  "only_update_on_change": false,
  "only_transmit_on_change": false,

  //Logging
  "logfile": "output.log"
  "log_level": "warning",
```

(continues on next page)

(continued from previous page)

```
"force_logging_flush": false,
"file_log_level": "",
"console_log_level": "",
"dump_log": false,

//Timing
"ignore_time_mismatch_warnings": false,
"uninterruptible": false,
"period": 0,
"offset": 0,
"time_delta": 0,
"minTimeDelta": 0,
"input_delay": 0,
"output_delay": 0,
"real_time": false,
"rt_tolerance": 0.2,
"rt_lag": 0.2,
"rt_lead": 0.2,
"grant_timeout": 0,
"wait_for_current_time_update": false,
"restrictive_time_policy": false,
"slow_responding": false,

//Iteration
"rollback": false,
"max_iterations": 10,
"forward_compute": false,

//Network
"interfaceNetwork": "local",
"brokeraddress": "127.0.0.1"
"reuse_address": false,
"noack": false,
"maxsize": 4096,
"maxcount": 256,
"networkretries": 5,
"osport": false,
"brokerinit": "",
"server_mode": "",
"interface": (local IP address),
"port": 1234,
"brokerport": 22608,
"localport": 8080,
"portstart": 22608,

"publications" | "subscriptions" | "inputs": [
{
  **"key": "publication key",**
  "type": "",
  "unit": "m",
```

(continues on next page)

(continued from previous page)

```

    "global": false,
    "connection_optional": true,
    "connection_required": false,
    "tolerance": -1,
    "targets": "",
    "buffer_data": false, indication the publication should buffer data
    "strict_input_type_checking": false,
    "alias": "",
    "ignore_unit_mismatch": false,
    "info": "",
  },
],
"publications" :[
  {
    "only_transmit_on_change": false,
  }
],
"subscriptions": [
  {
    "only_update_on_change": false,
  }
],
"inputs": [
  {
    "connections": 1,
    "input_priority_location": 0,
    "clear_priority_list": possible to have this as a config option?
    "single_connection_only": false,
    "multiple_connections_allowed": false
    "multi_input_handling_method": "average",
    "targets": ["pub1", "pub2"]
  }
],
"endpoints": [
  {
    "name": "endpoint name",
    "type": "endpoint type",
    "global": true,
    "destination" | "target" : "default endpoint destination",
    "alias": "",
    "subscriptions": "",
    "filters": "",
    "info": ""
  }
],
"filters": [
  {
    "name": "filter name",
    "source_targets": "endpoint name",
    "destination_targets": "endpoint name",
    "info": "",
    "operation": "randomdelay",
  }
]

```

(continues on next page)

(continued from previous page)

```
        "properties": {
            "name": "delay",
            "value": 600
        }
    }
]
```

3.3.3 General Federate Options

There are a number of flags which control how a federate acts with respect to timing and its signal interfaces.

name | -n (required)

API: `helicsFederateInfoSetCoreName` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Every federate must have a unique name across the entire federation; this is functionally the address of the federate and is used to determine where HELICS messages are sent. An error will be generated if the federate name is not unique.

core_type | coretype | coreType | -t ["zmq"]

API: `helicsFederateInfoSetCoreTypeFromString` [C++](#) | [C](#) | [Python](#) | [Julia](#)

There are a number of technologies or message buses that can be used to send HELICS messages among federates. Every HELICS enabled simulator has code in it that creates a core which connects to a HELICS broker using one of these messaging technologies. ZeroMQ (zmq) is the default core type and most commonly used but there are also cores that use TCP and UDP networking protocols directly (forgoing ZMQ's guarantee of delivery and reconnection functions), IPC (uses Boost's interprocess communication for fast in-memory message-passing but only works if all federates are running on the same physical computer), and MPI (for use on HPC clusters where MPI is installed). See the *User Guide page on core types* for more details.

core_name | corename | coreName []

API: `helicsFederateInfoSetCoreName` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Only applicable for `ipc` and `test` core types; otherwise can be left undefined.

core_init_string | coreinitstring | coreInitString | -i []

API: `helicsFederateInfoSetCoreInitString` C++ | C | Python | Julia

A command-line-like string that specifies options for the core as it connects to the federation. These options are:

- `--broker=` | `broker_address=` | `brokeraddress`: IP address of broker
- `--brokerport=`: Port number on which the broker is communicating
- `--broker_rank=`: For MPI cores only; identifies the MPI rank of the broker
- `--broker_tag=`: For MPI cores only; identifies the MPI tag of the broker
- `--localport=`: Port number to use when communicating with this core
- `--autobroker`: When included the core will automatically generate a broker
- `--key=`: Specifies a key to use when communicating with the broker. Only federates with this key specified will be able to talk to the broker with the same key value. This is used to prevent federations running on the same hardware from accidentally interfering with each other.
- `--profiler=log` - Send the profiling messages to the default logging file. `log` can be replaced with a path to an alternative file where only the profiling messages will be sent. See the [User Guide page on profiling](#) for further details.

In addition to these options, all options shown in the `broker_init_string` are also valid.

autobroker [false]

API: (none)

Automatically generate a broker if one cannot be connected to. For federations with only one broker (often the case) and/or with federations containing custom federates that were developed for this particular application, it can be convenient to create the broker in the process of creating a specific federate; this option allows that to take place. The downside to this is it creates a federation with a small amount of mystery as the broker is not clearly shown to be launched as its own federate alongside the other federates and those unfamiliar with the federation composition may have to spend some extra time to understand where the broker is coming from.

broker_init_string | brokerinitstring | brokerInitString [“”]

API: `helicsFederateInfoSetBrokerInitString` C++ | C | Python | Julia

String used to define the configuration of the broker if one is autogenerated. Such configuration typically includes things like broker IP addresses and port numbers. Again, if this is a co-simulation running on a single computer (and is the only HELICS co-simulation running on said computer) the default option is likely to be sufficient. The following options are available for this string:

- `--federates=` - The minimum number of federates expected to join a federation through the broker. Equivalent to `-f`.
- `--max_federates=` - The maximum number of federates allowed to join through a broker.
- `--name=` - Name of the broker; can be used by federates to specify which broker to use
- `--max_iterations=` - Maximum iterations allowed when using the re-iteration API

- `--min_broker_count=` - The minimum number of brokers that the co-simulation must have to begin initialization. (This option is not available for cores)
- `--max_broker_count=` - The maximum number of brokers that the co-simulation should allow. (This option is not available for cores)
- `--slow_responding` - Removes the requirement for the broker to respond to pings from other entities in the co-simulation in a timely manner and forces the assumption that this broker is still connected to the federation.
- `--restrictive_time_policy` - Forces the broker to use the most restrictive (conservative) timing policy when granting times to federates. Has the potential to increase co-simulation time as time grants may happen later than they actually need to.
- `--terminate_on_error` - All errors from any member of the federation will cause the broker to terminate the co-simulation for the entire federation.
- `--force_logging_flush` - Force writing to the log after every message.
- `--log_file=` - Name of file use for logging for this broker.
- `--log_level=` - Specifies the level of logging (both file and console) for this broker.
- `--file_log_level=` - Specifies the level of logging to file for this broker.
- `--console_log_level=` - Specifies the level of logging to file for this broker.
- `--dumplog` - Captures a record of all logging messages and writes them out to file or console when the broker terminates.
- `--tick=` - Heartbeat period in ms. When brokers fail to respond after 2 ticks secondary actions are taking to confirm the broker is still connected to the federation. Times can also be entered as strings such as “15s” or “75ms”.
- `--timeout=` milliseconds to wait for all the federates to connect to the broker (can also be entered as a time like ‘10s’ or ‘45ms’)
- `--network_timeout=` - Time to establish a socket connection in ms. Times can also be entered as strings such as “15s” or “75ms”.
- `--error_timeout=` - Time in ms to wait after an error state is reached before terminating. Times can also be entered as strings such as “15s” or “75ms”.
- `--query_timeout=` - Time in ms to wait for a query to complete. Times can also be entered as strings such as “15s” or “75ms”.
- `--grant_timeout=` - Time in ms to wait to allow a time request to wait before triggering diagnostic actions. Times can also be entered as strings such as “15s” or “75ms”.
- `--children=` - The minimum number of child objects the broker should expect before allowing entry to the initializing state.
- `--subbrokers=` - The minimum number of child objects the broker should expect before allowing entry to the initializing state. Same as `--children` but might be clearer in some cases with multilevel hierarchies.
- `--brokerkey=` - A broker key to use for connections to ensure federates are connecting with a specific broker and only appropriate federates connect with the broker. See [simultaneous co-simulations](#) for more information.
- `--profiler=log` - Send the profiling messages to the default logging file. `log` can be replaced with a path to an alternative file where only the profiling messages will be sent. See the [User Guide page on profiling](#) for further details.

terminate_on_error | terminateonerror | terminateOnError [false]

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_FLAG_TERMINATE_ON_ERROR [72]

If the `terminate_on_error` flag is set then a federate encountering an internal error will trigger a global error and cause the entire federation to terminate. Errors of this nature are typically the result of configuration errors, such as having a required publication that is not used or incompatible units or types on publications and subscriptions. This is the same option that is available in the `broker_init_string` but applied only to a specific federate (whereas when applied at the broker level it is effectively applied to all federates).

source_only | sourceonly | sourceOnly [false]

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_FLAG_SOURCE_ONLY [4]

Used to indicate to the federation that this federate is only producing data and has no inputs/subscriptions. Specifying this when appropriate allows HELICS to more efficiently grant times to the federation.

observer [false]

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_FLAG_OBSERVER [0]

Used to indicate to the federation that this federate produces no data and only has inputs/subscriptions. Specifying this when appropriate allows HELICS to more efficiently grant times to the federation.

3.3.4 Logging Options

log_file | logfile | logFile []

API: helicsFederateSetLogFile C++ | C | Python | Julia

Specifies the name of the log file where logging messages will be written.

log_level | loglevel | logLevel [0]

API: helicsFederateInfoSetIntegerProperty C++ | C | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_LOG_LEVEL [271]

Valid values:

- `no_print` - HELICS_LOG_LEVEL_NO_PRINT
- `error` - HELICS_LOG_LEVEL_ERROR
- `warning` - HELICS_LOG_LEVEL_WARNING

- `summary` - `HELICS_LOG_LEVEL_SUMMARY`
- `connections` - `HELICS_LOG_LEVEL_CONNECTIONS`
- `interfaces` - `HELICS_LOG_LEVEL_INTERFACES`
- `timing` - `HELICS_LOG_LEVEL_TIMING`
- `data` - `HELICS_LOG_LEVEL_DATA`
- `trace` - `HELICS_LOG_LEVEL_TRACE`

Determines the level of detail for log messages. In the list above, the keywords on the left can be used when specifying the logging level via a JSON configuration file. The enumerations on the right are used when configuring via the API.

`file_log_level` | `fileloglevel` | `fileLogLevel` [null]

API: `helicsFederateInfoSetIntegerProperty` C++ | C | Python | Julia

Property's enumerated name: `HELICS_PROPERTY_INT_FILE_LOG_LEVEL` [272]

Valid values: Same as in `loglevel`

Allows a distinct log level for the written log file to be specified. By default the logging level to file and console are identical and will only differ if `file_log_level` or `console_log_level` are defined.

`console_log_level` | `consoleloglevel` | `consoleLogLevel` [null]

API: `helicsFederateInfoSetIntegerProperty` C++ | C | Python | Julia

Property's enumerated name: `HELICS_PROPERTY_INT_CONSOLE_LOG_LEVEL` [274]

Valid values: Same as in `loglevel`

Allows a distinct log level for the written log file to be specified. By default the logging level to file and console are identical and will only differ if `file_log_level` or `console_log_level` are defined.

`force_logging_flush` | `forceloggingflush` | `forceLoggingFlush` [false]

API: `helicsFederateInfoSetFlagOption` C++ | C | Python | Julia

Property's enumerated name: `HELICS_FLAG_FORCE_LOGGING_FLUSH` [88]

Setting this option forces HELICS logging messages to be flushed to file after each one is written. This prevents the buffered IO most OSs implement to be bypassed such that all messages appear in the log file immediately after being written at the cost of slower simulation times due to more time spent writing to file.

dump_log | dumplog | dumpLog [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_DUMPLLOG` [89]

When set, a record of all messages is captured and written out to the log file at the conclusion of the co-simulation.

3.3.5 Timing Options

ignore_time_mismatch | ignoretimemismatch | ignoreTimeMismatch [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_IGNORE_TIME_MISMATCH_WARNINGS` [67]

If certain timing options (*i.e.* `period`, or `minTimeDelta`) are used it is possible for the time granted a federate to be greater than the requested time. This situation would normally generate a warning message, but if this flag is set those warnings are silenced.

uninterruptible [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_UNINTERRUPTIBLE` [1]

Normally, a federate will be granted a time earlier than it requested when it receives a message from another federate; the presence of any message implies there could be an action the federate needs to take and may generate new messages of its own. There are times, though, when it is important that the federate only be granted a time (and begin simulating/executing again) that it has previously requested. For example, there could be some controller that should only operate at fixed intervals even if new data arrives earlier. In these cases, setting the `uninterruptible` flag will prevent premature time grants.

period [0]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_TIME_PERIOD` [140]

Many time-based simulators have a minimum time-resolution or a user-configurable step size. The `period` parameter can be used to effectively synchronize the times that are granted with the defined simulation period. The default units for `period` are in seconds but the string for this parameter can include its own units (e.g. “2 ms” or “1 hour”). Setting `period` will force all time grants to occur at times of $n \times \text{period}$ even if subscriptions are updated, messages arrive, or the federate requests a time between periods. This value effectively makes the federates `uninterruptible` during the times between periods. Relatedly...

offset [0]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_TIME_OFFSET` [141]

There may be cases where it is preferable to have a simulator receive time grants that are offset slightly in time to one or more other federates. Defining an offset value allows this to take place; units are handled the same as in `period`. Setting both `period` and `offset`, will result in the all times granted to the federate in question being constrained to $n \cdot \text{period} + \text{offset}$.

time_delta | timeDelta | timedelta [1ns]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_TIME_DELTA` [137]

`timeDelta` has some similarities to `period`; where `period` constrained the granted time to regular intervals, `timeDelta` constrains the grant time to a minimum amount from the last granted time. Units are handled the same as in `period`.

input_delay | inputdelay | inputDelay [0]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_INPUT_TIME_DELAY` [148]

`inputDelay` specifies a delay in simulated time between when a signal arrives at a federate and when that federate is notified that a new value is available. `outputDelay` is similar but applies to signals being sent by a federate. Note that this applies to both value signals and message signals.

output_delay | outputdelay | outputDelay [0]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_TIME_PROPERTY_OUTPUT_TIME_DELAY` [150]

`outputDelay` is similar to `input_delay` but applies to signals being sent by a federate. Note that this applies to both value signals and message signals.

real_time | realtime | realTime [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_REALTIME` [16]

If set to true the federate uses `rt_lag` and `rt_lead` to match the time grants of a federate to the computer wall clock. If the federate is running faster than real time this will insert additional delays. If the federate is running slower than real time this will cause a force grant, which can lead to non-deterministic behavior. `rt_lag` can be set to `maxVal` to disable force grant

rt_lag | rtlag | rtLag [0.2] and rt_lead | rtlead | rtLead [0.2]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_TIME_RT_LAG` [143] and `HELICS_PROPERTY_TIME_RT_LEAD` [144]

Defines “real-time” for HELICS by setting tolerances for HELICS to use when running a real-time co-simulation. HELICS is forced to keep simulated time within this window of wall-clock time. Most general purpose OSes do not provide guarantees of execution timing and thus very small values of `rt_lag` and `rt_lead` (less than 0.005) are not likely to be achievable.

rt_tolerance | rttolerance | rtTolerance [0.2]

API: `helicsFederateInfoSetTimeProperty` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_PROPERTY_TIME_RT_TOLERANCE` [145]

Implements the same functionality of `rt_lag` and `rt_lead` but does so by using a single value to set symmetrical lead and lag constraints.

wait_for_current_time_update | waitforcurrenttimeupdate | waitForCurrentTimeUpdate [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE` [10]

If set to true, a federate will not be granted the requested time until all other federates have completed at least 1 iteration of the current time or have moved past it. If it is known that 1 federate depends on others in a non-cyclic fashion, this can be used to optimize the order of execution without iterating.

restrictive_time_policy | restrictivetimepolicy | restrictiveTimePolicy [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_RESTRICTIVE_TIME_POLICY` [11]

If set, a federate will not be granted the requested time until all other federates have completed at least 1 iteration of the current time or have moved past it. If it is known that 1 federate depends on others in a non-cyclic fashion, this can be used to optimize the order of execution without iterating.

Using the option `restrictive-time-policy` forces HELICS to use a fully conservative mode in granting time. This can be useful in situations beyond the current reach of the distributed time algorithms. It is generally used in cases where it is known that some federate is executing and will trigger someone else, but most federates won't know who that might be. This prevents extra messages from being sent and a potential for time skips. It is not needed if some federates are periodic and execute every time step. The flag can be used for federates, brokers, and cores to force very conservative timing with the potential loss of performance as well.

Only applicable to Named Input interfaces (see section on value federate interface types), if enabled this flag checks that data type of the incoming signals match that specified for the input.

slow_responding | slowresponding | slowResponding [false]

API: helicsFederateInfoSetFlagOption [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: HELICS_FLAG_SLOW_RESPONDING [29]

If specified on a federate, setting this flag indicates the federate may be slow in responding, and to not forcibly eject the federate from the federation for the slow response. This is an uncommon scenario.

If applied to a core or broker (`--slow_responding` in the `core_init_string` or `broker_init_string`, respectively), it is indicative that the broker doesn't respond to internal pings quickly and should not be disconnected from the federation for the slow response.

event_triggered [false]

API: helicsFederateInfoSetTimeProperty [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: HELICS_FLAG_EVENT_TRIGGERED [81]

For federates that are event-driven rather than timing driven, this flag must be set (to increase timing efficiency and avoid timing lock-ups). Event-driven federates are those that don't progress through simulation time at regular timesteps but instead wait for arriving messages to act. The most common examples are controller federates which generally request infinite time (well, `HELICS_TIME_MAXTIME`) and rely on HELICS to grant them an earlier time whenever a signal (often message) has arrived. Filter federates are another common federate type that must have this flag set.

3.3.6 Iteration

forward_compute | forwardcompute | forwardCompute [false]

API: helicsFederateInfoSetFlagOption [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: HELICS_FLAG_FORWARD_COMPUTE [14]

Indicates to the broker and the rest of the federation that this federate computes ahead of its granted time and can/does roll back when necessary. Federates able to do this (and who set this flag) allow more efficient time grants to the federation as a whole.

rollback [false]

API: helicsFederateInfoSetFlagOption [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: HELICS_FLAG_ROLLBACK [12]

Indicates to the broker and the rest of the federation that this federate can/does roll back when necessary. Federates able to do this (and who set this flag) allow more efficient time grants to the federation as a whole.

`max_iterations` | `maxiterations` | `maxIteration` [50]

API: `helicsFederateInfoSetIntegerProperty` C++ | C | Python | Julia

Property's enumerated name: `HELICS_PROPERTY_INT_MAX_ITERATIONS` [259]

For federates engaged in iteration (recomputing values based on updated inputs at a single simulation timestep) there may be a need to enforce a maximum number of iterations. This option allows that value to be set. When any federate reaches this number of iterations, HELICS will evaluate the federation as a whole and grant the next smallest time supported by the iterating federates. This time will only be granted to the federates that would be able to execute at this time.

3.3.7 General and Per Subscription, Input, or Publication

These options can be set globally for all subscriptions, inputs and publications for a given federate. Even after setting them globally, they can be included in the configuration for an individual subscription, input, or publication, over-riding the global setting.

`only_update_on_change` | `onlyupdateonchange` | `onlyUpdateOnChange` [false] and
`only_transmit_on_change` | `onlytransmitonchange` | `onlyTransmitOnChange` [false]

API: `helicsFederateInfoSetFlagOption` C++ | C | Python | Julia

Property's enumerated name: `HELICS_FLAG_ONLY_UPDATE_ON_CHANGE` [454] and
`HELICS_FLAG_ONLY_TRANSMIT_ON_CHANGE` [452]

Setting these flags prevents new value signals with the same value from being received by the federate or sent by the federate. Setting these flags will reduce the amount of traffic on the HELICS bus and can provide performance improvements in co-simulations with large numbers of messages.

tolerance

API: `helicsPublicationSetMinimumChange` and `helicsInputSetMinimumChange` C++ input and C++ publication | C input and C publication | Python input and Python publication | Julia input and Julia publication

This option allows the specific numerical definition of “change” when using the `only_update_on_change` and `only_transmit_on_change` options.

`connection_required` | `connectionrequired` | `connectionRequired` [false]

API: `helicsFederateInfoSetFlagOption` C++ | C | Python | Julia

Property's enumerated name: `HELICS_HANDLE_OPTION_CONNECTION_REQUIRED` [397]

When a federate is initialized, one of its tasks is to make sure the recipients of directed signals exist. If, after the federation is initialized, the recipient can't be found, then by default a warning is generated and written to the log file. If the `connections_required` flag is set, this warning becomes a fatal error that stops the co-simulation.

- `publications` - At least one federate must subscribe to the publications.
- `subscriptions` - The message being subscribed to must be provided by some other publisher in the federation.

`connection_optional` | `connectionoptional` | `connectionOptional` [false]

API: `helicsFederateInfoSetFlagOption` C++ | C | Python | Julia

Property's enumerated name: `HELICS_HANDLE_OPTION_CONNECITON_OPTIONAL` [402]

When an interface requests a target it tries to find a match in the federation. If it cannot find a match at the time the federation is initialized, then the default is to generate a warning. This will not halt the federation but will display a log message. If the `connections_optional` flag is set on a federate all subsequent `addTarget` calls on any interface will not generate any message if the target is not available.

3.3.8 Subscription, Input, and/or Publication Options

These options are valid for subscriptions, inputs, and/or publications (generically called “handles”). The APIs for dealing with registering these handles combine multiple options in the JSON config file and have varying levels of specificity (defining the date type for the handle or defining the handle as global). Rather than listing all APIs for the following options, the main APIs will be listed here and those using them can consult the API references to see which specific APIs are most applicable.

`helicsFederateRegisterPublication` C++ | C | Python | Julia

`helicsFederateRegisterSubscription` C++ | C | Python | Julia

`helicsFederateRegisterInput` C++ | C | Python | Julia

key (required)

- **publications** - The string in this field is the unique identifier (at the federate level) for the value that will be published to the federation. If `global` is set (see below) it must be unique to the entire federation.
- **subscriptions** - This string identifies the federation-unique value that this federate wishes to receive. Unless `global` has been set to `true` in the publishing's JSON configuration file, the name of the value is formatted as `<federate name>/<publication key>`. Both of these strings can be found in the publishing federate's JSON configuration file as the name and key strings, respectively. If `global` is `true` the string is just the key value.
- **input** - The string in this field is the unique identifier (at the federate level) that defines the input to receive value signals.

type [null]

HELICS supports data types and data type conversion (as best it can).

unit [null]

HELICS is able to do some levels of unit conversion, currently only on double type publications but more may be added in the future. The units can be any sort of unit string, a wide assortment is supported and can be compound units such as m/s^2 and the conversion will convert as long as things are convertible. The unit match is also checked for other types and an error if mismatching units are detected. A warning is also generated if the units are not understood and not matching. The unit checking and conversion is only active if both the publication and subscription specify units.

global [false]

(publications only) `global` is used to indicate that the value in `key` will be used as a global name when other federates are subscribing to the message. This requires that the user ensure that the name is used only once across all federates. Setting `global` to `true` is handy for federations with a small number of federates and a small number of message exchanges as it allows the `key` string to be short and simple. For larger federations, it is likely to be easier to set the flag to `false` and accept the extra naming.

buffer_data | bufferdata | bufferData [false]

API: `helicsInputSetOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_HANDLE_OPTION_BUFFER_DATA` [411]

(only valid for inputs and subscriptions) Setting this flag will buffer the last value sent during the initialization phase of HELICS co-simulations. When the execution phase begins, that value will be resent to the receiving handle.

ignore_units_mismatch | ignoreunitmismatch | ignoreUnitMismatch [null]

API: Under normal operation, handles that are connected (value signals flowing between them) are required to have units that either match or can be directly converted between. If mismatching units are connected, an error is thrown; when this flag is set that error is suppressed.

info [“”]

API: `helicsInputSetInfo` [C++](#) | [C](#) | [Python](#) | [Julia](#) The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

strict_input_type_checking | strictinputtypechecking | strictInputTypeChecking [false]

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_STRICT_TYPE_CHECKING [414]

When an interface requests a target it tries to find a match in the federation. If it cannot find a match at the time the federation is initialized, then the default is to generate a warning. This will not halt the federation but will display a log message. If the `connections_optional` flag is set on a federate all subsequent `addTarget` calls on any interface will not generate any message if the target is not available.x

3.3.9 Publication-only Options

targets

C++ | C | Python | Julia Used to specify which inputs should receive the values from this output. This can be a list of output keys/names.

3.3.10 Input-only Options

Inputs can receive values from multiple sending handles and the means by which those multiple data points for a single handle are managed can be specified with several options. See the *User Guide entry* for further details.

targets

C++ | C | Python | Julia Inputs can specify which outputs (typically publications) they should be pulling from. This is similar to subscriptions but inputs can allow multiple outputs to feed to the same input. This can be a list of output keys/names.

connections []

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_CONNECTIONS [522]

Allows an integer number of connections to be considered value for this input handle. Similar to `multiple_connections_allowed` but allows the number of sending handles to be defined to a specific number.

input_priority_location | inputprioritylocation | inputPriorityLocation []

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_INPUT_PRIORITY_LOCATION [510]

When receiving values from multiple sending handles, when the values are received they or organized as a vector. This option is used to define which value in that vector has priority. The API can be called multiple times to set successive priorities.

clear_priority_list | clearprioritylist | clearPriorityList [false]

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_CLEAR_PRIORITY_LIST [512]

When receiving values from multiple sending handles, when the values are received they or organized as a vector. This option is used to clear that priority list and redefine which values have priority.

single_connection_only | singleconnectiononly | singleConnectionOnly [false]

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_SINGLE_CONNECTION_ONLY [407] When set, this forces the input handle to have only one sending handle it will receive from. Setting this flag serves as a sort of double-check to ensure that only one other handle is sending to this input handle and that the federation has been configured as expected.

multiple_connections_allowed | multipleconnectionsallowed | multipleConnectionsAllowed [true]

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_MULTIPLE_CONNECTIONS_ALLOWED [409]

When set, this flag allows the input handle to receive values from multiple other handles.

multi_input_handling_method | multiinputhandlingmethod | multiInputHandlingMethod [none]

API: helicsInputSetOption C++ | C | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_MULTI_INPUT_HANDLING_METHOD [507] Property values:

- none | no_op
- or
- sum
- max
- min
- average
- mean
- vectorize
- diff

Given that an input can have multiple data sources, a method of reducing those multiple values into one needs to be defined. HELICS supports a number of mathematical operation to perform this reduction.

3.3.11 Endpoint Options

As in the value handles, the registration of endpoints is done through a single API that incorporates multiple options. And as in the value handles, there is a `global` API option to allow the name of the endpoint to be considered global to the federation.

API: `helicsFederateRegisterEndpoint` [C++](#) | [C](#) | [Python](#) | [Julia](#)

name (required)

The name of the endpoint as it will be known to the rest of the federation.

type []

API: (none)

HELICS supports data types and data type conversion (as best it can).

destination | target []

API: `helicsEndpointSetDefaultDestination` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Defines the default destination for a message sent from this endpoint.

alias []

API: (none)

Creates a local alias for a handle that may have a long name.

subscriptions []

API: `helicsEndpointSubscribe` [C++](#) | [C](#) | [Python](#) | [Julia](#)

filters [null]

See section on Filter Options.

info [“”]

API: `helicsEndpointSetInfo` [C++](#) | [C](#) | [Python](#) | [Julia](#) The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

3.3.12 Filter Options

Filters are registered with the core or through the application API. There are also Filter object that hide some of the API calls in a slightly nicer interface. Generally a filter will define a target endpoint as either a source filter or destination filter. Source filters can be chained, as in there can be more than one of them. At present there can only be a single non-cloning destination filter attached to an endpoint.

Non-cloning filters can modify the message in some ways, cloning filters just copy the message and may send it to multiple destinations.

On creation, filters have a target endpoint and an optional name. Custom filters may have input and output types associated with them. This is used for chaining and automatic ordering of filters. Filters do not have to be defined on the same core as the endpoint, and in fact can be anywhere in the federation, any messages will be automatically routed appropriately.

API: `helicsFederateRegisterFilter` ([C++](#) | [C](#) | [Python](#) | [Julia](#)) to create/register the filter and then `helicsFilterAddSourceTarget` ([C++](#) | [C](#) | [Python](#) | [Julia](#)) or `helicsFilterAddDestinationTarget` ([C++](#) | [C](#) | [Python](#) | [Julia](#)) to associate it with a specific endpoint

name []

API: (none)

Name of the filter; must be unique to a federate.

source_targets, sourcetargets, sourceTargets []

API: `` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Acts on previously registered filter and associated with a specific endpoint of the federate.

destination_targets, destinationtargets, destinationTargets []

API: `` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Acts on previously registered filter and associated with a specific endpoint of the federate.

info [“”]

API: helicsFilterSetInfo [C++](#) | [C](#) | [Python](#) | [Julia](#) The info field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

operation []

API: helicsFederateRegisterFilter [C++](#) | [C](#) | [Python](#) | [Julia](#)

Filters have a predefined set of operations they can perform. The following list defines the valid operations for filters. Most filters require additional specifications in properties data structure, an example of which is shown for each filter type.

reroute

This filter reroutes a message to a new destination. it also has an optional filtering mechanism that will only reroute if some patterns are matching. The patterns should be specified by “condition” in the set string the conditions are regular expression pattern matching strings.

Example property object:

```
...
  "operation": "reroute",
  "properties": [
    {
      "name": "newdestination",
      "value": "endpoint name"
    },
    {
      "name": "condition",
      "value": "regular expression string"
    }
  ]
...
```

delay

This filter will delay a message by a certain amount fo time.

Example property object:

```
...
  "operation": "delay",
  "properties": {
    "name": "delay",
    "value": "76 ms",
  },
...
```

random_delay | randomdelay | randomDelay

This filter will randomly delay a message according to specified random distribution available options include distribution selection, and 2 parameters for the distribution some distributions only take one parameter in which case the second is ignored. The distributions available are based on those available in the C++ [random](#) library.

- **constant**
 - param1="value" this just generates a constant value
- **uniform**
 - param1="min"
 - param2="max"
- **bernoulli** - the bernoulli distribution will return param2 if the bernoulli trial returns true, 0.0 otherwise. Param1 is the probability of returning param2
 - param1="prob"
 - param2="value"
- **binomial**
 - param1=t (cast to int)
 - param2="p"
- **geometric**
 - param1="prob" the output is param2*geom(param1) so multiplies the integer output of the geometric distribution by param2 to get discrete chunks
- **poisson**
 - param1="mean"
- **exponential**
 - param1="lambda"
- **gamma**
 - param1="alpha"
 - param2="beta"
- **weibull**
 - param1="a"
 - param2="b"
- **extreme_value**
 - param1="a"
 - param2="b"
- **normal**
 - param1="mean"
 - param2="stddev"
- **lognormal**
 - param1="mean"

- param2="stddev"
- **chi_squared**
 - param1="n"
- **cauchy**
 - param1="a"
 - param2="b"
- **fisher_f**
 - param1="m"
 - param2="n"
- **student_t**
 - param1="n"

```
...
  "operation": "randomdelay",
  "properties": [
    {
      "name": "distribution",
      "value": normal
    },
    {
      "name": "mean",
      "value": 0
    },
    {
      "name": "stdev",
      "value": 1
    }
  ],
...

```

random_drop | randomdrop | randomDrop

This filter will randomly drop a message, the drop probability is specified, and is modeled as a uniform distribution between zero and one.

```
...
  "operation": "random_drop",
  "properties": {
    "name": "prob",
    "value": 0.5,
  },
...

```

clone

This filter will copy a message and send it to the original destination plus a new one.

```
...
  "operation": "clone",
  "properties": {
    "name": "add delivery",
    "value": "endpoint name",
  },
  ...
```

3.3.13 Profiling

HELICS 2.8 and v3.0.1 have a profiling capability that allows users to measure the time spent waiting for HELICS to grant it time and how much time is spent executing native code. These measurements are the foundation to understanding how to improve computation performance in a federation. Further details are provided in the [Profiling page in the User Guide](#). When enabling profiling at the federate level there are a few APIs that can be utilized.

profiling [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_PROFILING` [93]

Setting this flag enables profiling for the federate.

profiler [“”]

Turns on profiling for the federate and allows the specification of the log file where profiling messages will be written. No API is possible with this option as it must be specified prior to the creation of the federates.

local_profiling_capture [false]

API: `helicsFederateInfoSetFlagOption` [C++](#) | [C](#) | [Python](#) | [Julia](#)

Property's enumerated name: `HELICS_FLAG_LOCAL_PROFILING_CAPTURE` [96]

Setting this flag sends the profiling messages to the local federate log rather than propagating them up to the core and/or broker.

profiling_marker [false]

API: `helicsFederateInfoSetFlagOption` C++ | C | Python | Julia

Property's enumerated name: `HELICS_FLAG_PROFILING_MARKER` [95]

Generates an additional `marker` message if logging is enabled.

3.3.14 Network

For most HELICS users, most of the time, the following network options are not needed. They are most likely to be needed when working in complex networking environments, particularly when running co-simulations across multiple sites with differing network configurations. Many of these options require non-trivial knowledge of network operations and rather and it is assumed that those that needs these options will understand what they do, even with the minimal descriptions given.

interface network

API:

See multiple options for `-local`, `-ipv4`, `-ipv6`, `-all`

reuse_address | reuseaddress | reuseAddress [false]

API: (none)

Allows the server to reuse a bound address, mostly useful for tcp cores.

noack_connect | noackconnect | noackConnect [false]

Specify that a `connection_ack` message is not required to be connected with a broker.

max_size | maxsize | maxSize[4096]

API: (none)

Message buffer size. Can be increased for large messages to limit the number of retries by the underlying networking protocols.

max_count | maxcount | maxCount [256]

API: (none)

Maximum number of messages in queue. Can be increased for large volumes of messages to limit the number of retries by the underlying networking protocols.

network_retries | networkretries | networkRetries [5]

API: (none) Maximum number of network retry attempts.

use_os_port | useosport | useOsPort [false]

API: (none) Setting this flag specifies that the OS should set the port for the HELICS message bus. HELICS will ask the operating system which port to use and will use the indicated port.

client or server [null]

API: (none) specify that the network connection should be a server or client. By default neither option is enabled.

local_interface | localinterface | localInterface [local address]

API: (none) the local interface to use for the receive ports.

port | -p []

API: (none) Port number to use.

broker_port | brokerport | brokerPort []

API: (none)

The port to use to connect to the broker.

`broker_name` | `brokername` | `brokerName` []

API: (none)

`local_port` | `localport` | `localPort` []

API: (none) port number for the local receive port.

`port_start` | `portstart` | `portStart` []

API: (none) starting port for automatic port definitions.

3.4 API Reference

3.4.1 C++ API (doxygen)

3.4.2 C API

3.4.3 Python API

3.4.4 Julia API

QUICK LINKS

- *Configuration Option Reference*
- *Queries*
- *Environment Variables*
- *C function Reference*
- *CMake Variables*
- *HELICS Apps*