Integrated Plasma Simulator (IPS) Documentation

Release 0.8.1+5.g6f02737.dirty

UT-Battelle, LLC

Jan 27, 2023

CONTENTS

IPS (Integrated Plasma Simulator) is an environment to orchestrate complex coupled simulation workflows on parallel computers. The IPS is designed primarily for use in a batch-processing environment, with a batch job typically comprising a single invocation of the framework, calling the individual physics codes many times as the simulation progresses.

Contents:

CHAPTER

ONE

INTRODUCTION

Welcome to the documentation for the Integrated Plasma Simulator (IPS). The documents contained here will provide information regarding obtaining, using and developing the IPS and some associated tools.

The IPS was originally developed for the SWIM project and is designed for coupling plasma physics codes to simulate the interactions of various heating methods on plasmas in a tokamak. The physics goal of the project is to better understand how the heating changes the properties of the plasma and how these heating methods can be used to improve the stability of plasmas for fusion energy production.

The IPS framework is thus designed to couple standalone codes flexibly and easily using python wrappers and file-based data coupling. These activities are not inherently plasma physics related and the IPS framework can be considered a general code coupling framework. The framework provides services to manage:

- the orchestration of the simulation through component invocation, task launch and asynchronous event notification mechanisms,
- configuration of complex simulations using familiar syntax,
- file communication mechanisms for shared and internal (to a component) data, as well as checkpoint and restart capabilities,

The framework performs the task, configuration, file and resource management, along with the event service, to provide these features.

1.1 Where to Start?

For those who have never run the IPS before, you should start with *Getting Started*. It starts from the beginning with how to obtain the IPS code, build and run some sample simulations on two different platforms.

The *User Guides* section has documents on basic and advanced user topics. For those who have used the IPS before or have done the tutorial and are ready to create their own run, the *Reference Guide for Running IPS Simulations* document walks you through the process of using the IPS to examine a computational or physics problem, with practical hints on what to consider through out the preparation, running and analysis/debugging processes. Additional documentation for basic simulation construction include *The Configuration File - Explained*. *The IPS for Driver and Component Developers* provides component developers with basic information on the construction of a component and integrating it into the IPS, guidance on how to construct drivers and IPS services API reference. Additional documents on advanced topics such as multiple levels of parallelism, computational considerations, fault tolerance and performance analysis are located in the *User Guides* chapter.

Developers of the IPS framework and services, or brave souls who wish to understand how these pieces work, should look at the *code listings*. The code listings here will include internal and external APIs. The developer guides include information about the design of the IPS at a high level and the framework and managers at a lower level to acquaint developers with the structures and mechanisms that are used in the IPS framework source code.

1.2 Acknowledgments

This documentation has been primarily written or adapted from other sources by Samantha Foley, as part of the SWIM team. Don Batchelor provided examples and documentation that provided the basis for the *Getting Started* and *Basic IPS Usage* sections. Wael Elwasif provided much of the code documentation and initial documents on the directory structure and build process.

CHAPTER

TWO

GETTING STARTED

This document will guide you through the process of running an IPS simulation and describe the overall structure of the IPS. It is designed to help you build and run your first IPS simulation. It will serve as a tutorial on how to get, build, and run your first IPS simulation, but not serve as a general reference for constructing and running IPS simulations. See the *Basic User Guides* for a handy reference on running and constructing simulations in general, and for more in-depth explanations of how and why the IPS works.

Warning: The were major changes in IPS from the old (up to July 2020) way of doing things to a new way. See *Migrating from old IPS v0.1.0 to new IPS*.

2.1 Obtaining, Dependencies, Platforms

The IPS code is currently located on the GitHub repository. In order to checkout a copy, you must have git installed on the machine you will be using. Once you have git you can check out the IPS thusly:

```
git clone https://github.com/HPC-SimTools/IPS-framework.git
```

2.1.1 Dependencies

The IPS framework is written in Python, and requires Python 3.6+. There are a few other packages that may be needed for certain components or utilities. The framework does use the Python package ConfigObj, and urllib3 is used to communicate with the *IPS Portal*.

2.1.2 Other Utilities

Resource Usage Simulator (RUS) This is a utility for simulation the execution of tasks in the IPS for research purposes.

Requires: Matplotlib (which requires Numpy/Scipy)

Warning: The RUS (Resource Usage Simulator) has not been updated to python 3 or for the changes in IPS and will not function in it current state.

Documentation The documentation you are reading now was created by a Python-based tool called Sphinx.

Requires: Sphinx

Plus anything that the components or underlying codes that you are using need (e.g., MPI, math libraries, compilers). For the example in this tutorial, all packages that are needed are already available on the target machines and the shell configuration script sets up your environment to use them.

2.2 Building and Setting up Your Environment

IPS has two dependencies (ConfigObj and urllib3) required to run in addition to python version 3.6. The dependencies will be automatically installed when you install `ipsframework`. There is an optional dependency Dask that will enable Dask to be used for task pool scheduling, see *`submit_tasks()`*. IPS is tested to work with dask and distributed 2.5.2 but may work with earlier versions.

It is recommend to install IPS in an conda environment, see *Create and install in conda environment*.

It can be simply installed with

```
python -m pip install ipsframework
```

The latest development version of IPS can be installed directly from github with pip

```
python -m pip install git+https://github.com/HPC-SimTools/IPS-framework.git
```

You can install a specific version by, for examples version `v0.3.0`

```
python -m pip install ipsframework==0.3.0
# or
python -m pip install git+https://github.com/HPC-SimTools/IPS-framework.git@v0.3.0
```

Otherwise you can download the source code and install from there.

2.2.1 Installing IPS from source

The source code can be cloned with git from https://github.com/HPC-SimTools/IPS-framework with either ssh:

```
git clone git@github.com:HPC-SimTools/IPS-framework.git
```

of over https:

```
git clone https://github.com/HPC-SimTools/IPS-framework.git
```

Install in current python environment, from within the IPS-framework source directory

```
python -m pip install .
```

If you are using the system python and don't want to install as root you can do a user only install with

```
python -m pip install --user .
```

`ips.py` should now be installed in your `PATH` and you should be able to run `ips.py --config=simulation.config --platform=platform.conf`

Install in editable mode

Installing in editable allows you to modify the source code and use it in from you source directory without reinstalling. This doesn't actually install the package but creates an egg link.

```
python -m pip install -e .
```

Note: You may need to use `pip3` and `python3` if you default `python` is not `python3`.

2.2.2 Create and install in conda environment

Note: For specific instruction on setting up conda environments on NERSC set *Installing IPS on NERSC*.

First you need conda, you can either install the full Anaconda package or Minconda (recommenced) which is a minimal installer for conda.

First create a conda environment and activate it, this environment is named `ips`. You can use any version of python >= 3.6

```
conda create -n ips python=3.9
conda activate ips
```

Next install IPS into this environment in the same way as above. *e.g.*

```
python -m pip install ipsframework
```

And you are good to go.

To leave your conda environment

```
conda deactivate
```

Installing packages

To see which packages are currently install in your environment run:

```
conda list
```

You can install just the packages you need by

```
conda install dask matplotlib netcdf4 ...
```

CHAPTER

THREE

USER GUIDES

This directory has all of the user guides for using the IPS (see the component and portal user guides for further information pertaining to those topics). It is organized in a series of *basic IPS usage* topics and *advanced IPS usage* topics, both are chock-full of examples and skeletons.

How do I know if I am a Basic or Advanced user? Basic IPS usage documents contain information that is intended for those who have run a few simulations and need a refresher on how to set up and run an existing simulation. These documents will help users run or make small modifications to existing simulations, including ways the IPS and other utilities can be used to examine scientific problems.

Advanced IPS usage documents contain information for *writers* of drivers and components. These documents will help those who wish to make new components and drivers, make significant changes to an existing component or driver, examine the performance of the IPS and components, or those who would like to understand how to use the multiple levels of parallelism and asynchronous communication mechanisms effectively.

Basic IPS Usage

Introduction to the IPS A handy reference for constructing and running applications, this document helps users through the process of running a simulation based on existing components. It also includes: terminology, examples, and guidance on how to translate a computational or scientific question into a series of IPS runs.

***The Configuration File - Explained*:** Annotated version of the configuration file with explanations of how and why the values are used in the framework and components.

***Platform Configuration File - Explained*:** Annotated platform configuration file and explanation of the manual allocation specification interface.

Advanced IPS Usage

***The IPS for Driver and Component Developers*:** This guide contains the elements of components and drivers, suggestions on how to construct a simulation, how to add the new component to a simulation and the repository, as well as, an IPS services guide to have handy when writing components and drivers. This guide is for components and drivers based on the *generic driver* model. More sophisticated logic and execution models are covered in the following document.

Create a component package This guide shows an example of creating a separate component package, which depends on the ipsframework and can be installed into you python environment. This is also an example of using *MODULE* instead of *SCRIPT* in the component configuration section.

Migration from old IPS to new IPS A guide on converting from the old (up to July 2020) way of doing things to the new way.

Setting up environment on NERSC How to setup conda environments on NERSC for using IPS.

Using the IPS Portal How to setup simulation to use the IPS portal.

3.1 Reference Guide for Running IPS Simulations

This reference guide is designed to help you through the process of setting up a simulation to run. It provides instructions on how to change configuration files and how to build and run the IPS on a given platform, as well as, determine if the simulation is setup correctly and will produce the correct data. In the various sections the user will find a series of questions designed to help the user plan for the preparation, execution, and post-processing of a run (or series of runs).

3.1.1 Terminology

Before going further, some basic definitions of terms that are used in the IPS must be presented. These terms are specific to the IPS and may be used in other contexts with different meanings. These are brief definitions and designed to remind the user of their meaning.

3.1.2 Elements of a Simulation

Head node The *head node* is how this documentation refers to any login, service or head node that acts as the gateway to a cluster or MPP. It is where the Python codes and some helper scripts run, including the framework, services and components.

Compute node A *compute node* is a node that exists in the compute partition of a parallel machine. It is designed for running compute intensive and parallel applications.

Batch allocation The *batch allocation* is the set of (compute) nodes given to the framework by the system's scheduler. The framework services manage the allocation of resources and launching of tasks on compute nodes within this allocation.

Framework The *framework* serves as the structure that contains the components, drivers and services for the simulation(s). It provides the infrastructure for the different elements to interact. It is the piece of software that is executed, and uses the services to invoke, run and manage the drivers, components, and tasks.

Component A *component* is a Python class that interacts with other components (typically the driver) and tasks using the services. A *physics* component typically uses the Python class to adapt a standalone physics code to be coupled with other components. Logically, each component contributes something to the simulation at hand, whether it is a framework functionality, like a bridge to the portal, or a model of some physical phenomena, like RF heating sources.

Task A *task* is an executable that runs on compute node(s) launched by the services on behalf of the component. These executables are the ones who do the heavy physics computation and dominate the run time, allowing the Python components and framework to manage the orchestration and other services involved in managing a multiphysics simulation. Most often tasks are parallel codes using MPI for interprocess communication.

Driver (Component) The *driver* is a special component in that it is the first one to be executed for the simulation. It is responsible for invoking its constituent components, implementing the time stepping and other logic among components, and global data operations, such as checkpointing.

Init (Component) The *init* component is a special in that it is invoked by the framework and is the first one to be executed for the simulation. It is responsible for performing any initialization needed by the driver before it begins its execution cycle.

Port A *port* is a category of component that can be implemented by different component implementations, i.e., components that wrap codes that different mathematical models of the same phenomenon. Each component that has the same port must implement the same interface (i.e., implement functions with the same names - in the IPS all components implement "init", "step", and "finalize"), and provide the same functionality in a coupled simulation. In most cases, this means that it updates the same values in the plasma state. Drivers use the port name of a component to obtain a reference for that component at run time, as specified in the configuration file.

Services The framework *services* provide APIs for setting up the simulation, and managing data, resources, tasks, component invocations, access to configuration data and communication via an event service during execution. For more details, see *code listings*. Component writers should check out the *services API* for relevant services and tips on how to use them.

Data files Each component specifies the input and output *data files* it needs for a given simulation. These file names and locations are used to stage data in and out for each time step. Note that these are not the same as the *plasma state files*, in that *data files* are component local (and thus private).

Plasma State files The *plasma state* is a utility and set of files that allow multiple components to contribute values to a set of files representing the shared data about the plasma. These shared files are specified in the configuration file and access is managed through the framework services data management API. Component writers may need to write scripts to translate between plasma state files and the files expected/generated by the underlying executable.

Configuration file The *configuration file* allows the user to describe how a simulation is to be run. It uses a third-party Python package called ConfigObj to easily parse the shell-like, hierarchical syntax. In the configuration file there are sections describing the following aspects of the simulation. They are all explained in further detail in *The Configuration File - Explained*.

Platform Configuration file The *platform configuration file* contains platform specific information needed by the framework for task and resource management, as well as paths needed by the portal and configuration manager. These rarely change, so the version in the top level of the IPS corresponding to the platform you are running on should be used.

Batch script The *batch script* tells the batch scheduler how and what to run, including the number of processes and nodes for the allocation, the command to launch the IPS, and any other information that the batch scheduler needs to know to run your job.

3.1.3 Sample workflow

This section consists of an outline of how the IPS is intended to be used. It will walk you through the steps from forming an idea of what to run, through running it and analyzing the results. This will also serve as a reference for running IPS simulations. If you are not comfortable with the elements of an IPS simulation, then you should start with the sample simulations in *Getting Started* and review the terminology above.

Problem Formation

Before embarking on a simulation experiment, the problem that you are addressing needs to be determined. The problem may be a computational one where you are trying to determine if a component works properly, or an experiment to determine the scalability or sensitivity to computation parameters, such as time step length or number of particles. The problem may pertain to a study of how a component, or set of components, compare to previous results or real data. The problem may be to figure out for a set of variations which one produces the most stable plasma conditions. In each case, you will need to determine:

- what components are needed to perform this experiment?
- what input files must be obtained, prepared or generated (for each component and the simulation as a whole)?
- does this set of components make sense?
- what driver(s) are needed to perform this experiment?
- do new components and drivers need to be created?
- does it make sense to run multiple simulations in a single IPS instance?
- how will multiple simulations effect the computational needs and amount of data that is produced?
- what plasma state files are needed?

- where will initial plasma state values (and those not modeled by components in this scenario) come from?
- how much compute time and resources are needed for each task? the simulation as a whole?
- are there any restrictions on where or when this experiment can be run?
- how will the output data be analyzed?
- where will the output data go when the simulation is completed?
- when and where will the output data be analyzed?

Once you have a plan for constructing, managing and analyzing the results of your simulation(s), it is time to begin preparation.

A Brief Introduction to Writing and Modifying Components

In many cases, new components or modifications to existing components need to be made. In this section, the anatomy of a component and a driver are explained for a simple invocation style of execution. (see *Advanced User Guide* for more information on creating components and drivers with complex logic, parallelism and asynchronous control flow).

Each component is derived from the `Component` class, meaning that each IPS component inherits a few base capabilities, and then must augment them. Each IPS component must implement the following function bodies for the component class:

`init(self, timeStamp=0)` This function performs pre-simulation setup activities such as reading in global configuration parameters, checking configuration parameters, updating input files and internal state. (Component configuration parameters are populated *before* `init` is ever called.)

`step(self, timeStamp=0)` This function is the main part of the component. It is responsible for launching any tasks, and managing the input, output and plasma state during the course of the step.

`finalize(self, timeStamp=0)` This function is called after the simulation has completed and performs any clean up that is required by the component. Typically there is nothing to do.

`checkpoint(self, timeStamp=0)` This function performs a checkpoint for the component. All of the files marked as restart files in the configuration file are automatically staged to the checkpoint area. If the component has any internal knowledge or logic, or if there are any additional files that are needed to restart, this should be done explicitly here.

`restart(self, timeStamp=0)` This function replaces `init` when restarting a simulation from a previous simulation step. It should read in data from the appropriate files and set up the component so that it is ready to compute the next step.

To create a new component, there are two ways to do it, start from "scratch" by copying and renaming the skeleton component (`skeleton_comp.py`) to your desired location, or by modifying an existing component (e.g., `example_comp.py`). When creating your new component, keep in mind that it should be somewhat general and usable in multiple contexts. In general, for things that change often, you will want to use component configuration variables or input files to drive the logic or set parameters for the tasks. For more in depth information about how to create components and add them to the build process, see *Developing Drivers and Components for IPS Simulations*.

When changing an existing component that will diverge from the existing version, be sure to create a new version. If you are editing an existing component to make it better, be sure to document what you changexs.

Setup Simulation

At this point, all components and drivers should be added to the repository, and any makefiles modified or created (see *makefile section* of component writing guide). You are now ready to set up the execution environment, build the IPS, and prepare the input and configuration files.

Execution Environment

First, the platform on which to run the simulation must be determined. When choosing a platform, take in to consideration:

- The parallelism of the tasks you are running
  - Does your problem require 10s, 100s or 1000s of cores?
  - How well do your tasks take advantage of "many-core" nodes?
- The location of the input files and executables
  - Does your input data exist on a suitable platform?
  - Is it reasonable to move the data to another machine?
- Time and CPU hours
  - How much time will it take to run the set of simulations for the problem?
  - Is there enough CPU time on the machine you want to use?
- Dealing with results
  - Do you have access to enough hard drive space to store the output of the simulation until you have the time to analyze and condense it?

Once you have chosen a suitable platform, you may install IPS, see *Building and Setting up Your Environment*.

Second, construct input files or edit the appropriate ones for your simulation. This step is highly dependent on your simulation, but make sure that you check for the following things (and recheck after constructing the configuration file!):

- Does each component have all the input files it needs?
- Are there any global initial files, and are they present? (This includes any plasma state and non-plasma state files.)
- For each component input file: Are the values present, valid, and consistent?
- For the collection of files for each component: Are the values present, valid, and consistent?
- For the collection of files for each simulation: Are the values present, valid, and consistent?
- Do the components model all of the targeted domain and phenomena of the experiment?
- Does the driver use the components you expect?
- Does the driver implement the data dependencies between the components as you wish?

Third, you must construct the configuration file. It is helpful to start with a configuration file that is related to the experiment you are working on, or you may start from the example configuration file, and edit it from there. Some configuration file values are user specific, some are platform specific, and others are simulation or component specific. It may be helpful to save your personal versions on each machine in your home directory or some other persistent storage location for reuse and editing. These tend not to be good files to keep in subversion, however there are some examples in the example directory to get you started. The most common and required configuration file entries are explained here. For more a more complete description of the configuration options, see *The Configuration File - Explained*.

- User Data Section:

```
USER_W3_DIR = <location of your web directory on this platform>
USER_W3_BASEURL = <URL of your space on the portal>
USER = <user name>          # Optional, if missing the unix username is used
```

Set these values to the www directory you created for your own runs, a matching url for the portal to store your run info, and your user name (this is used on the portal to identify simulations you run). These should be the same for all of your runs on a given platform.

- Simulation Info Section:

```
RUN_ID = <short name of run>
TOKAMAK_ID = <name of the tokamak>
SHOT_NUMBER = 1
...
SIM_NAME = ${RUN_ID}_${SHOT_NUMBER}

OUTPUT_PREFIX =

SIM_ROOT = <location of output tree>

RUN_COMMENT = <used by portal to help identify what ran and why>
TAG = <grouping string>
...
SIMULATION_MODE = NORMAL
RESTART_TIME =
RESTART_ROOT = ${SIM_ROOT}
```

In this section the simulation is described and key locations are specified. *RUN_COMMENT* and *TAG*, along with *RUN_ID*, *TOKAMAK_ID*, and *SHOT_NUMBER* are used by the portal to describe this simulation. *RUN_ID*, *TOKAMAK_ID*, and *SHOT_NUMBER* are commonly used to construct the *SIM_NAME*, which is often used in as the directory name of the *SIM_ROOT*. And finally, the *SIMULATION_MODE* and related items identify the simulation as a *NORMAL* or *RESTART* run.

- Logging Section:

```
LOG_FILE = ${RUN_ID}_sim.log
LOG_LEVEL = DEBUG | WARN | INFO | CRITICAL
```

The logging section defines the name of the log file and the default level of logging for the simulation. The log file for the simulation will contain all logging messages generated by the components in this simulation. Logging messages from the framework and services will be written to the framework log file. The *LOG_LEVEL* may be the following and may differ from the framework log level (in order of most verbose to least)[1]:

  - *DEBUG* - all messages are produced, including debugging messages to help diagnose problems. Use this setting for debugging runs only.
  - *INFO* - these are messages stating what is happening, as opposed to what is going wrong. Use this logging level to get an idea of how the different pieces of the simulation interact, without extraneous messages from the debugging level.
  - *WARN* - these messages are produced when the framework or component expects different conditions, but has an alternative behavior or default value that is also valid. In most cases these messages are harmless, but may indicate a behavior that is different than expected. This is the most common logging level.

[1] For more information and guidance about how the Python logging module works, see the Python logging module tutorial.

  - *ERROR* - conditions that throw exceptions typically also produce an error message through the logging mechanism, however not all errors result in the failure of a component or the framework.
  - *CRITICAL* - only messages about fatal errors are produced. Use this level when using a well known and reliable simulation.

- Plasma State Section:

```
STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state

# Config variables defining simulation specific names for plasma state files
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
CURRENT_CQL = ${SIM_NAME}_ps_CQL.dat
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSDSK = ${SIM_NAME}_ps.jso

# List of files that constitute the plasma state
STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
STATE_FILES2 = ${CURRENT_CQL} ${CURRENT_DQL} ${CURRENT_JSDSK}
STATE_FILES = ${STATE_FILES1} ${STATE_FILES2}
```

  Specifies the naming convention for the plasma state files so the framework and components can manipulate and reference them in the config file and during execution. The initial file locations are also specified here.

- Ports Section:

```
[PORTS]
    NAMES = INIT DRIVER MONITOR EPA RF_IC NB FUS

# Required ports - DRIVER and INIT
    [[DRIVER]]
        IMPLEMENTATION = GENERIC_DRIVER

    [[INIT]]
        IMPLEMENTATION = minimal_state_init
# Physics ports
   [[RF_IC]]
       IMPLEMENTATION = model_RF_IC

   [[FP]]
       IMPLEMENTATION = minority_model_FP

   [[FUS]]
       IMPLEMENTATION = model_FUS

   [[NB]]
       IMPLEMENTATION = model_NB

   [[EPA]]
       IMPLEMENTATION = model_EPA

   [[MONITOR]]
```

(continues on next page)

(continued from previous page)

```
        IMPLEMENTATION = monitor_comp_4
```

The ports section specifies which ports are included in the simulation and which implementation of the port is to be used. Note that a *DRIVER* must be specified, and a warning will be issued if there is no *INIT* component present at start up. The value of *IMPLEMENTATION* for a given port *must* correspond to a component description below.

- Component Configuration Section:

```
[<component name>]
    CLASS = <port name>
    SUB_CLASS = <type of component>
    NAME = <class name of component implementation>
    NPROC = <# of procs for task invocations>
    BIN_PATH = <location of binaries>
    INPUT_DIR = ${DATA_TREE_ROOT}/<location of input directory>
        INPUT_FILES = <input files for each step>
        OUTPUT_FILES = <output files to be archived>
        STATE_FILES = ${CURRENT_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
        RESTART_FILES = ${INPUT_FILES} <extra state files>
    SCRIPT = ${BIN_PATH}/<component implementation>
    MODULE = <module name to use instead of script e.g. package.component>
```

For each component, fill in or modify the entry to match the locations of the input, output, plasma state, and script locations. Also, be sure to check the *NPROC* entry to suit the problem size and scalability of the executable, and add any component specific entries that the component implementation calls for. It allows multiple users to access the same data and have reasonable assurance that they are indeed using the same versions. The plasma state files must be part of the simulation plasma state. It may be a subset if there are files that are not needed by the component on each step. Additional component-specific entries can also appear here to signal a piece of logic or set a data value.

- Checkpoint Section:

```
[CHECKPOINT]
   MODE = WALLTIME_REGULAR
   WALLTIME_INTERVAL = 15
   NUM_CHECKPOINT = 2
   PROTECT_FREQUENCY = 5
```

This section specifies the checkpoint policy you would like enforced for this simulation, and the corresponding parameters to control the frequency and number of checkpoints taken. See the comments in the same configuration file or the configuration file *documentation*. If you are debugging or running a component or simulation for the first time, it is a good idea to take frequent checkpoints until you are confident that the simulation will run properly.

- Time Loop Section:

```
[TIME_LOOP]
    MODE = REGULAR
    START = 0.0
    FINISH = 20.0
    NSTEP = 5
```

This section sets up the time loop to help the driver manage the time progression of the simulation. If you are debugging or running a component or simulation for the first time, it is a good idea to take very few steps until you are confident that the simulation will run properly.

Lastly, double-check that your input files and config file are both self-consistent and make physics sense.

Run Simulation

Now, that you have everything set up, it is time to construct the batch script to launch the IPS. Just like the configuration files, this is something that tends to be user specific and platform specific, so it is a good idea to keep local copy in a persistent directory on each platform you tend to use for easy modification.

As an example, here is a skeleton of a batch script:

```
#! /bin/bash
#PBS -A <project code for accounting>
#PBS -N <name of simulation>
#PBS -j oe                          # joins stdout and stderr
#PBS -l walltime=0:6:00
#PBS -l mppwidth=<number of *cores* needed>
#PBS -q <queue to submit job to>
#PBS -S /bin/bash
#PBS -V

ips.py [--config=<config file>]+ \
        --platform=platform.conf \
        --log=<name of log file> \
       [--debug]  \
       [--nodes=<number of nodes in this allocation>] \
       [--ppn=<number of processes per node for this allocation>]
```

Note that you can only run one instance of the IPS per batch submission, however you may run multiple simulations in the same batch allocation by specifying multiple entries on the command line as a comma-separated list, e.g., `--config=<config file 1>, <config file 2>`. Each config file must have a unique file name, and *SIM_ROOT*. The different simulations will share the resources in the allocation, in many cases improving the resource efficiency, however this may make the execution time of each individual simulation a bit longer due to waiting on resources.

The IPS also needs information about the platform it is running on (`--platform=platform.conf`) and a log file (`--logfile=<name of log file>`)for the framework output. Platform files for commonly used platforms are provided in the top-level of the ips directory. It is strongly recommended that you use the appropriate one for launching IPS runs. See *Platforms and Platform Configuration* for more information on how to use or create these files.

Lastly, there are some optional command line arguments that you may use. `--debug` will turn on debugging information from the framework. `--nodes` and `--ppn` allow the user to manually set the number of nodes and processes per node for the framework. This will override any detection by the framework and should be used with caution. It is, however, a convenient way to run the ips on a machine without a batch scheduler.

Analysis and/or Debugging

Once your run (or set of runs) is done, it is time to look at the output. First, we will examine the structure of the output tree:

${SIM_ROOT}/

${PORTAL_RUNID}

File containing the portal run ids that are associated with this directory. There can be more than one.

<platform config file>

<simulation configuration files>

Each simulation configuration file that used this sim root.

restart/

<each checkpoint>/

<each component>/

Directory containing the restart files for this checkpoint

simulation_log/

Directory containing the event log for each runid.

simulation_results/

<each time step>/

components/

<each component>/

Directory containing the output files for the given component at the given step.

<each component>/

Directory containing the output files for each step. File names are appended with the time step to avoid collisions.

simulation_setup/

<each component>/

Directory containing the input files from the beginning of the simulation.

work/

<each component>/

Directory where the component computes from time step to time step. Left-over input and output files from the last step will be present at the end of the simulation.

There are a few tools for visualizing (and light analysis) of a run or set of runs:

- Portal web interface to PCMF: This tool is a web interface to the PCMF tool (see below). It has recently been integrated into the portal for quick and remote viewing. For more in depth analysis, viewing and printing of graphs from the monitor component, use the more powerful standalone version of PCMF.
- PCMF: A tool to Plot and Compare multiple Monitor Files (`ips/components/monitor/monitor_4/PCMF.py`) is the local Python version of the web tool. It uses Matplotlib to generate plots of the different values in the plasma state over the course of the simulation. It also allows you to generate graphs for more than one set of monitor files. Examples and instructions are located in the repo and are coming soon to this documentation.
- ELVis: This tool graphs values from netCDF (plasma state) files through a web browser plugin or using the Java client.

Using these utilities, your own scripts or manual inspection results can be analyzed, or bugs found. Debugging a coupled simulation is more complicated than debugging a standalone code. Here are some things to consider when a problem is encountered:

- Problems using the framework

  - Was an exception thrown? If so, what was it and where did it come from? If you don't understand the exception, talk to a framework developer.
  - Was something missing in the configuration file?
  - Were the components invoked and tasks launched as expected?
  - Did you use the proper implementation of the component and executable?
  - Was your compute environment/permissions/batch allocation set up properly?
- Data between components
  - Does each component update all the values in the plasma state it needs to?
  - Does each component update all output files it uses internally properly?
  - Are the components updating the plasma state in the right order?
- Physics code problem
  - Did a task return an error code?
  - Does the component check for a bad return code and handle it properly?
  - Is the code that is launched have the proper command line arguments?
  - Are the input and output files properly adapted to the executable?
  - Does the executable fail in standalone mode?
  - Was the executable built properly?
  - Were all necessary input and source files found?

If you are working out a problem, it is always good to:

- Turn on debugging output using the `--debug` flag on the command line, and setting the LOG_LEVEL in the configuration file to DEBUG.
- Turn on debugging output in physics codes to see what is going on during each task.
- Use frequent checkpoints to restart close to where the problem starts.
- Reduce the number of time steps to the minimum needed to produce the problem.
- Only change one thing before rerunning the simulation to determine what fixes the problem.

3.2 The Configuration File - Explained

This section will detail the different sections and fields of the configuration file and how they relate to a simulation. The configuration file is designed to let the user to easily set data items used by the framework, components, tasks, and the portal from run to run. There are user specific, platform specific, and component specific entries that need to be modified or verified before running the IPS in the given configuration. After a short overview of the syntax of the package used by the framework to make sense of the configuration file, a detailed explanation of each line of the configuration file is presented.

3.2.1 Syntax and the ConfigObj module

ConfigObj is a Python package for reading and writing config files. The syntax is similar to shell syntax (e.g., use of $ to reference variables), uses square brackets to create named sections and nested subsections, comma-separated lists and comments indicated by a "#".

In the example configuration file below, curly braces ({}) are used to clarify references to variables with underscores (_). Any left-hand side value can be used as a variable after it is defined. Additionally, any platform configuration value can be referenced as a variable in the configuration file as well.

3.2.2 Configuration File - Line by Line

Platform Configuration Override Section It is possible for the configuration file to override entries in the platform configuration file. It is rare and users should use caution when overriding these values. See *Platform Configuration File - Explained* for details on these values.

```
#HOST =
#MPIRUN =
#NODE_ALLOCATION_MODE =
```

User Data Section

The following items are specific to the user and should be changed accordingly. They will help you to identify your runs in the portal (*USER*), and also store the data from your runs in particular web-enabled locations for post-processing (*USER_W3_DIR* on the local machine, *USER_W3_BASEURL* on the portal). All of the items in this section are optional.

```
USER_W3_DIR = /project/projectdirs/m876/www/ssfoley
USER_W3_BASEURL = http://portal.nersc.gov/project/m876/ssfoley
USER = ssfoley                # Optional, if missing the unix username is used
```

Simulation Information Section These items describe this configuration and is used for describing and locating its output, information for the portal, and location of the source code of the IPS.

** Mandatory items: *SIM_ROOT*, *SIM_NAME*, *LOG_FILE*

RUN_ID, *TOKOMAK_ID*, *SHOT_NUMBER* - identifiers for the simulation that are helpful for SWIM users. They are often used to form a hierarchical name for the simulation, identifying related runs.

OUTPUT_PREFIX - used to prevent collisions and overwriting of different simulations using the same *SIM_ROOT*.

SIM_NAME - used to identify the simulation on the portal, and often to name the output tree.

LOG_FILE - name of the log file for this simulation. The framework log file is specified at the command line.

LOG_LEVEL - sets the logging level for the simulation. If empty, the framework log level is used, which defaults to *WARNING*. See *Logging* for details on the logging capabilities in the IPS. Possible values: *DEBUG*, *INFO*, *WARNING*, *ERROR*, *EXCEPTION*, *CRITICAL*.

SIM_ROOT - location of output tree. This directory will be created if it does not exist. If the directory already exists, then data files will be added, possibly overwriting existing data.

```
RUN_ID = Model_seq              # Identifier for this simulation run
TOKAMAK_ID = ITER
SHOT_NUMBER = 1               # Identifier for specific case for this tokamak
                              # (should be character integer)
```

(continues on next page)

(continued from previous page)

```
SIM_NAME = ${RUN_ID}_${TOKAMAK_ID}_${SHOT_NUMBER}

OUTPUT_PREFIX =
LOG_FILE = ${RUN_ID}_sim.log
LOG_LEVEL = DEBUG             # Default = WARNING

# Simulation root - path of the simulation directory that will be constructed
# by the framework
SIM_ROOT = /scratch/scratchdirs/ssfoley/seq_example

# Description of the simulation for the portal
SIMULATION_DESCRIPTION = sequential model simulation using generic driver.py
RUN_COMMENT = sequential model simulation using generic driver.py
TAG = sequential_model                        # for grouping related runs
```

Simulation Mode

This section describes the mode in which to run the simulation. All values are optional.

SIMULATION_MODE - describes whether the simulation is starting from *init* (*NORMAL*) or restarting from a checkpoint (*RESTART*). The default is *NORMAL*. For RESTART, a restart time and directory must be specified. These values are used by the driver to control how the simulation is initialized. *RESTART_TIME* must coincide with a checkpoint save time. *RESTART_DIRECTORY* may be $SIM_ROOT if there is an existing current simulation there, and the new work will be appended, such that it looks like a seamless simulation.

NODE_ALLOCATION_MODE - sets the default execution mode for tasks in this simulation. If the value is *EXCLUSIVE*, then tasks are assigned whole nodes. If the value is *SHARED*, sub-node allocation is used so tasks can shared nodes thus using the allocation more efficiently. It is the users responsibility to understand how node sharing will impact the performance of their tasks.

```
SIMULATION_MODE = NORMAL   # NORMAL | RESTART
RESTART_TIME = 12         # time step to restart from
RESTART_ROOT = ${SIM_ROOT}
NODE_ALLOCATION_MODE = EXCLUSIVE # SHARED | EXCLUSIVE
```

Plasma State Section

The locations and names of the plasma state files are specified here, along with the directory where the global plasma state files are located in the simulation tree. It is common to specify groups of plasma state files for use in the component configuration sections. These files should contain all the shared data values for the simulation so that they can be managed by the driver.

```
STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state

# Config variables defining simulation specific names for plasma state files
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
CURRENT_CQL = ${SIM_NAME}_ps_CQL.dat
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSDSK = ${SIM_NAME}_ps.jso

# List of files that constitute the plasma state
```

(continues on next page)

(continued from previous page)

```
STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE}  ${NEXT_STATE} ${CURRENT_EQDSK}
STATE_FILES2 = ${CURRENT_CQL} ${CURRENT_DQL} ${CURRENT_JSDSK}
STATE_FILES = ${STATE_FILES1} ${STATE_FILES2}
```

Ports Section

The ports section identifies which ports and their associated implementations that are to be used for this simulation. The ports section is defined by `[PORTS]`. *NAMES* is a list of port names, where each needs to appear as a subsection (e.g., `[[DRIVER]]`). Each port definition section must contain the entry *IMPLEMENTATION* whose value is the name of a component definition section. These are case sensitive names and should be named such that someone familiar the components of this project has an understanding of what is being modeled. The only mandatory port is *DRIVER*. It should be named *DRIVER*, but the implementation can be anything, as long as it is defined. If no *INIT* port is defined, then the framework will produce a warning to that effect. There may be more port definitions than listed in *NAMES*.

```
[PORTS]
   NAMES = INIT DRIVER MONITOR EPA RF_IC NB FUS

# Required ports - DRIVER and INIT
   [[DRIVER]]
      IMPLEMENTATION = GENERIC_DRIVER

   [[INIT]]
      IMPLEMENTATION = minimal_state_init

# Physics ports

  [[RF_IC]]
      IMPLEMENTATION = model_RF_IC

  [[FP]]
      IMPLEMENTATION = minority_model_FP

  [[FUS]]
      IMPLEMENTATION = model_FUS

  [[NB]]
      IMPLEMENTATION = model_NB

  [[EPA]]
      IMPLEMENTATION = model_EPA

  [[MONITOR]]
      IMPLEMENTATION = monitor_comp_4
```

Component Configuration Section

Component definition and configuration is done in this "section." Each component configuration section is defined as a section (e.g., `[model_RF_IC]`). Each entry in the component configuration section is available to the component at runtime using that name (e.g., *self.NPROC*), thus these values can be used to create specific simulation cases using generic components. Variables defined within a component configuration section are local to that section, but values may be defined in terms of the simulation values defined above (e.g., *STATE_FILES*).

** Mandatory entries: *SCRIPT*, *NAME*, *BIN_PATH*, *INPUT_DIR*

CLASS - commonly this is the port name or the first directory name in the path to the component implementation in

`ips/components/`.

SUB_CLASS - commonly this is the name of the code or method used to model this port, or the second directory name in the path to the component implementation in `ips/components/`.

NAME - name of the class in the Python script that implements this component.

MODULE - module name to use instead of script e.g. `package.component`, see *Create a component package* for an example.

NPROC - number of processes on which to launch tasks.

BIN_PATH - path to script and any other helper scripts and binaries. This is used by the framework and component to find and execute helper scripts and binaries.

BINARY - the binary to launch as a task. Typically, these binaries are found in the

PHYS_BIN or some subdirectory therein. Otherwise, you can make your own variable and put the directory where the binary is located there.

INPUT_DIR - directory where the input files (listed below) are found. This is used during initialization to copy the input files to the work directory of the component.

INPUT_FILES - list of files (relative to *INPUT_DIR*) that need to be copied to the component work directory on initialization. *OUTPUT_FILES* - list of output files that are produced that need to be protected and archived on a call to `services.ServicesProxy.stage_output_files()`.

STATE_FILES - list of plasma state files used and modified by this component. If not present, then the files specified in the simulation entry *STATE_FILES* is used.

RESTART_FILES - list of files that need to be archived as the checkpoint of this component.

NODE_ALLOCATION_MODE - sets the default execution mode for tasks in this component. If the value is *EXCLUSIVE*, then tasks are assigned whole nodes. If the value is *SHARED*, sub-node allocation is used so tasks can share nodes thus using the allocation more efficiently. If no value or entry is present, the simulation value for *NODE_ALLOCATION_MODE* is used. It is the users responsibility to understand how node sharing will impact the performance of their tasks. This can be overridden using the *whole_nodes* and *whole_sockets* arguments to `services.ServicesProxy.launch_task()`.

Additional values that are specific to the component may be added as needed, for example certain data values like *PPN*, paths to and names of other executables used by the component or alternate *NPROC* values are examples. It is the responsibility of the component writer to make sure users know what values are required by the component and what the valid values are for each.

```
[model_EPA]
    CLASS = epa
    SUB_CLASS = model_epa
    NAME = model_EPA
    NPROC = 1
    BIN_PATH = /path/to/bin
    INPUT_DIR = ${DATA_TREE_ROOT}/model_epa/ITER/hy040510/t20.0
        INPUT_STATE_FILE = hy040510_002_ps_epa__tsc_4_20.000.cdf
        INPUT_EQDSK_FILE = hy040510_002_ps_epa__tsc_4_20.000.geq
        INPUT_FILES = model_epa_input.nml ${INPUT_STATE_FILE} ${INPUT_EQDSK_FILE}
        OUTPUT_FILES = internal_state_data.nml
        STATE_FILES = ${CURRENT_STATE} ${NEXT_STATE} ${CURRENT_EQDSK}
        RESTART_FILES = ${INPUT_FILES} internal_state_data.nml
    SCRIPT = ${BIN_PATH}/model_epa_ps_file_init.py

[monitor_comp_4]
```

(continues on next page)

(continued from previous page)

```
    CLASS = monitor
    SUB_CLASS =
    NAME = monitor
    NPROC = 1
    W3_DIR = ${USER_W3_DIR}              # Note this is user specific
    W3_BASEURL = ${USER_W3_BASEURL}      # Note this is user specific
    TEMPLATE_FILE= basic_time_traces.xml
    BIN_PATH = /path/to/bin
    INPUT_DIR = /path/to/components/monitor/monitor_4
    INPUT_FILES = basic_time_traces.xml
    OUTPUT_FILES = monitor_file.nc
    STATE_FILES = ${CURRENT_STATE}
    RESTART_FILES = ${INPUT_FILES} monitor_restart monitor_file.nc
    SCRIPT = ${BIN_PATH}/monitor_comp.py
```

Checkpoint Section

This section describes when checkpoints should be taken by the simulation. Drivers should be written such that at the end of each step there is a call to `services.ServicesProxy.checkpoint_components()`. This way the services use the settings in this section to either take a checkpoint or not.

Selectively checkpoint components in *comp_id_list* based on the configuration section *CHECKPOINT*. If *Force* is `True`, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is `True`, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to `False`.

The *CHECKPOINT_MODE* option controls determines if the components checkpoint methods are invoked. Possible *MODE* options are:

WALLTIME_REGULAR: checkpoints are saved upon invocation of the service call `checkpoint_components()`, when a time interval greater than, or equal to, the value of the configuration parameter WALLTIME_INTERVAL had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to `checkpoint_components()` before WALLTIME_INTERVAL seconds have passed since the last successful checkpoint result in a NOOP.

WALLTIME_EXPLICIT: checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable WALLTIME_VALUES. Let [t_0, t_1, . . . , t_n] be the list of wall clock time values specified in the configuration parameter WALLTIME_VALUES. Then checkpoint(T) = True if T >= t_j, for some j in [0,n] and there is no other time T_1, with T > T_1 >= T_j such that checkpoint(T_1) = True. If the test fails, the call results in a NOOP.

PHYSTIME_REGULAR: checkpoints are saved at regularly spaced "physics time" intervals, specified in the configuration parameter PHYSTIME_INTERVAL. Let PHYSTIME_INTERVAL = PTI, and the physics time stamp argument in the call to checkpoint_components() be pts_i, with i = 0, 1, 2, . . . Then checkpoint(pts_i) = True if pts_i >= n PTI , for some n in 1, 2, 3, . . . and pts_i - pts_prev >= PTI, where checkpoint(pts_prev) = True and pts_prev = max (pts_0, pts_1, ..pts_i-1). If the test fails, the call results in a NOOP.

PHYSTIME_EXPLICIT: checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable PHYSTIME_VALUES. Let [pt_0, pt_1, . . . , pt_n] be the list of physics time values specified in the configuration parameter PHYSTIME_VALUES. Then checkpoint(pt) = True if pt >= pt_j, for some j in [0,n] and there is no other physics time pt_k, with pt > pt_k >= pt_j such that checkpoint(pt_k) = True. If the test fails, the call results in a NOOP.

The configuration parameter NUM_CHECKPOINT controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of NUM_CHECKPOINT are:

- NUM_CHECKPOINT = n, with n > 0 –> Keep the most recent n checkpoints
- NUM_CHECKPOINT = 0 –> No checkpoints are made/kept (except when *Force* = `True`)

- NUM_CHECKPOINT < 0 –> Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

```
[CHECKPOINT]
   MODE = WALLTIME_REGULAR
   WALLTIME_INTERVAL = 15
   NUM_CHECKPOINT = 2
   PROTECT_FREQUENCY = 5
```

Time Loop Section

The time loop specifies how time progresses for the simulation in the driver. It is not required by the framework, but may be required by the driver. Most simulations use the time loop section to specify the number and frequency of time steps for the simulation as opposed to hard coding it into the driver. It is a helpful tool to control the runtime of each step and the overall simulation. It can also be helpful when looking at a small portion of time in the simulation for debugging purposes.

MODE - defines the following entries. If mode is *REGULAR – START*, *FINISH* and *NSTEP* are used to generate a list of times of length *NSTEP* starting at *START* and ending at *FINISH*. If mode is *EXPLICIT – VALUES* contains the (whitespace separated) list of times that are are to be modeled.

```
[TIME_LOOP]
    MODE = REGULAR
    START = 0.0
    FINISH = 20.0
    NSTEP = 5
```

3.3 Platforms and Platform Configuration

This section will describe key aspects of the platforms that the IPS has been ported to, key locations relevant to the IPS, and the platform configuration settings in general and specific to the platforms described below.

Important Note - while this documentation is intended to remain up to date, it may not always reflect the current status of the machines. If you run into problems, check that the information below is accurate by looking at the websites for the machine. If you are still having problems, contact the framework developers.

3.3.1 Ported Platforms

Each subsection will contain information about the platform in question. If you are porting the IPS to a new platform, these are the items that you will need to know or files and directories to create in order to port the IPS. You will also need a platform configuration file (*described below*). Available queue names are listed with the most common ones in **bold**.

The platforms below fall into the following categories:

- general production machines - large production machines on which the majority of runs (particularly production runs) are made.
- experimental systems - production or shared machines that are being used by a subset of SWIM members for specific research projects. These systems may also be difficult for others to get accounts.
- formerly used systems - machines that the IPS was ported to but we either do not have time on that machine, it has been retired by its hosting site, or it is not in wide use anymore.
- single user systems - laptop or desktop machines for testing small problems.

General Production

Cori

Cori is a Cray XC40 managed by NERSC.

- Account: You must have an account at NERSC and be added to the Atom project's group (atom) to log on and access the set of physics binaries in the *PHYS_BIN*.
- Logging on - `ssh cori.nersc.gov -l <username>`
- Architecture - 2,388 Haswell nodes, 32 cores per node, 128GB memory per node + 9,668 KNL nodes, 68 cores per node, 96 GB memory
- Environment:
  - OS - SUSE Linux Enterprise Server 15 (SLES15)
  - Batch scheduler/Resource Manager - Slurm
  - Queues - **debug**, **regular**, premium, interactive, ...
  - Parallel Launcher (e.g., mpirun) - srun
  - Node Allocation policy - exclusive or shared node allocation
- Project directory - `/global/project/projectdirs/atom`
- Data Tree - `/global/common/software/atom/cori/data`
- Physics Binaries - `/global/common/software/atom/cori/binaries`
- WWW Root - `/global/project/projectdirs/atom/www/<username>`
- WWW Base URL - `http://portal.nersc.gov/project/atom/<username>`

Retired/Formerly Used Systems

Franklin

Franklin is a Cray XT4 managed by NERSC.

- Account: You must have an account at NERSC and be added to the SWIM project's group (m876) to log on and access the set of physics binaries in the *PHYS_BIN*.
- Logging on - `ssh franklin.nersc.gov -l <username>`
- Architecture - 9,572 nodes, 4 cores per node, 8 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - Queues - **debug**, **regular**, low, premium, interactive, xfer, iotask, special
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - `/project/projectdirs/m876/`
- Data Tree - `/project/projectdirs/m876/data/`

- Physics Binaries - `/project/projectdirs/m876/phys-bin/phys/`
- WWW Root - `/project/projectdirs/m876/www/<username>`
- WWW Base URL - `http://portal.nersc.gov/project/m876/<username>`

Hopper

Hopper is a Cray XE6 managed by NERSC.

- Account: You must have an account at NERSC and be added to the SWIM project's group (m876) to log on and access the set of physics binaries in the *PHYS_BIN*.
- Logging on - `ssh hopper.nersc.gov -l <username>`
- Architecture - 6384 nodes, 24 cores per node, 32 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - Queues - **debug**, **regular**, low, premium, interactive
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - `/project/projectdirs/m876/`
- Data Tree - `/project/projectdirs/m876/data/`
- Physics Binaries - `/project/projectdirs/m876/phys-bin/phys/`
- WWW Root - `/project/projectdirs/m876/www/<username>`
- WWW Base URL - `http://portal.nersc.gov/project/m876/<username>`

Stix

Stix is a SMP hosted at PPPL.

- Account: You must have an account at PPPL to access their Beowulf systems.
- Logging on:
  1. Log on to the PPPL vpn (https://vpn.pppl.gov)
  2. `ssh <username>@portal.pppl.gov`
  3. `ssh portalr5`
- Architecture - 80 cores, 440 GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - PBS (Torque), Moab
  - Queues - **smpq** (this is how you specify that you want to run your job on stix)
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)

- Project directory - `/p/swim1/`
- Data Tree - `/p/swim1/data/`
- Physics Binaries - `/p/swim1/phys/`
- WWW Root - `/p/swim/w3_html/<username>`
- WWW Base URL - `http://w3.pppl.gov/swim/<username>`

Viz/Mhd

Viz/mhd are SMP machines hosted at PPPL. These systems appear not to be online any more.

- Account: You must have an account at PPPL to access their Beowulf systems.
- Logging on:
  1. Log on to the PPPL vpn (https://vpn.pppl.gov)
  2. `ssh <username>@portal.pppl.gov`
- Architecture - ? cores, ? GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - PBS (Torque), Moab
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)
- Project directory - `/p/swim1/`
- Data Tree - `/p/swim1/data/`
- Physics Binaries - `/p/swim1/phys/`
- WWW Root - `/p/swim/w3_html/<username>`
- WWW Base URL - `http://w3.pppl.gov/swim/<username>`

Pingo

Pingo was a Cray XT5 hosted at ARSC.

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture - 432 nodes, 8 cores per node, ? memory per node
- Environment:
  - OS - ?
  - Batch scheduler/Resource Manager - ?
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - ?

- Data Tree - ?
- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

Jaguar

Jaguar is a Cray XT5 managed by OLCF.

- Account: You must have an account for the OLCF and be added to the SWIM project group for accounting and files sharing purposes, if we have time on this machine.
- Logging on - `ssh jaguar.ornl.gov -l <username>`
- Architecture - 13,688 nodes, 12 cores per node, 16 GB memory per node
- Environment:
  - OS - Cray Linux Environment (CLE)
  - Batch scheduler/Resource Manager - PBS, Moab
  - Queues - debug, production
  - Parallel Launcher (e.g., mpirun) - aprun
  - Node Allocation policy - exclusive node allocation
- Project directory - ?
- Data Tree - ?
- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

Experimental Systems

Swim

Swim is a SMP hosted by the fusion theory group at ORNL.

- Account: You must have an account at ORNL and be given an account on the machine.
- Logging on - `ssh swim.ornl.gov -l <username>`
- Architecture - ? cores, ? GB memory
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - None
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed (whole machine looks like one node)
- Project directory - None

- Data Tree - None
- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

Pacman

Pacman is a linux cluster hosted at ARSC.

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture:
  - 88 nodes, 16 cores per node, 64 GB per node
  - 44 nodes, 12 cores per node, 32 GB per node
- Environment:
  - OS - Red Hat Linux 5.6
  - Batch scheduler/Resource Manager - Torque (PBS), Moab
  - Queues - debug, standard, standard_12, standard_16, bigmem, gpu, background, shared, transfer
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI?)
  - Node Allocation policy - node sharing allowed
- Project directory - ?
- Data Tree - ?
- Physics Binaries - ?
- WWW Root - ?
- WWW Base URL - ?

Iter

Iter is a linux cluster (?) that is hosted ???.

- Account: You must have an account to log on and use the system.
- Logging on - ?
- Architecture - ? nodes, ? cores per node, ? GB memory per node
- Environment:
  - OS - linux
  - Batch scheduler/Resource Manager - ?
  - Queues - ?
  - Parallel Launcher (e.g., mpirun) - mpiexec (MPICH2)
  - Node Allocation policy - node sharing allowed

- Project directory - `/project/projectdirs/m876/`
- Data Tree - `/project/projectdirs/m876/data/`
- Physics Binaries - `/project/projectdirs/m876/phys-bin/phys/`
- WWW Root - ?
- WWW Base URL - ?

Odin

Odin is a linux cluster hosted at Indiana University.

- Account: You must have an account to log on and use the system.
- Logging on - `ssh odin.cs.indiana.edu -l <username>`
- Architecture - 128 nodes, 4 cores per node, ? GB memory per node
- Environment:
  - OS - GNU/Linux
  - Batch scheduler/Resource Manager - Slurm, Maui
  - Queues - there is only one queue and it does not need to specified in the batchscript
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed
- Project directory - None
- Data Tree - None
- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

Sif

Sif is a linux cluster hosted at Indiana University.

- Account: You must have an account to log on and use the system.
- Logging on - `ssh sif.cs.indiana.edu -l <username>`
- Architecture - 8 nodes, 8 cores per node, ? GB memory per node
- Environment:
  - OS - GNU/Linux
  - Batch scheduler/Resource Manager - Slurm, Maui
  - Queues - there is only one queue and it does not need to specified in the batchscript
  - Parallel Launcher (e.g., mpirun) - mpirun (OpenMPI)
  - Node Allocation policy - node sharing allowed
- Project directory - None

- Data Tree - None
- Physics Binaries - None
- WWW Root - None
- WWW Base URL - None

Single User Systems

The IPS can be run on your laptop or desktop. Many of the items above are not present or relevant in a laptop/desktop environment. See the next section for a sample platform configuration settings.

3.3.2 Platform Configuration File

The platform configuration file contains platform specific information that the framework needs. Typically it does not need to be changed for one user to another or one run to another (except for manual specification of allocation resources). For *most* of the platforms above, you will find platform configuration files of the form `<machine name>.conf`. It is not likely that you will need to change this file, but it is described here for users working on experimental machines, manual specification of resources, and users who need to port the IPS to a new machine.

```
HOST = cori
MPIRUN = srun

########################################
# resource detection method
########################################

NODE_DETECTION = slurm_env # checkjob | qstat | pbs_env | slurm_env

########################################
# node topology description
########################################

PROCS_PER_NODE = 32
CORES_PER_NODE = 32
SOCKETS_PER_NODE = 1

########################################
# framework setting for node allocation
########################################
# MUST ADHERE TO THE PLATFORM'S CAPABILITIES
#   * EXCLUSIVE : only one task per node
#   * SHARED : multiple tasks may share a node
# For single node jobs, this can be overridden allowing multiple
# tasks per node.

NODE_ALLOCATION_MODE = EXCLUSIVE # SHARED | EXCLUSIVE
USE_ACCURATE_NODES = ON
```

HOST name of the platform. Used by the portal.

MPIRUN command to launch parallel applications. Used by the task manager to launch parallel tasks on compute nodes. If you would like to launch a task directly without the parallel launcher (say, on a SMP style machine or workstation), set this to "eval" – it tells the task manager to directly launch the task as `<binary> <args>`.

NODE_DETECTION method to use to detect the number of nodes and processes in the allocation. If the value is "manual," then the manual allocation description is used. If nothing is specified, all of the methods are attempted and the first one to succeed will be used. Note, if the allocation detection fails, the framework will abort, killing the job.

TOTAL_PROCS number of processes in the allocation[3].

NODES number of nodes in the allocation[?].

PROCS_PER_NODE number of processes per node (ppn) for the framework[2].

CORES_PER_NODE number of cores per node[1].

SOCKETS_PER_NODE number of sockets per node[?].

NODE_ALLOCATION_MODE 'EXCLUSIVE' for one task per node, and 'SHARED' if more than one task can share a node[?]. Simulations, components and tasks can set their node usage allocation policies in the configuration file and on task launch.

GPUS_PER_NODE number of GPUs per node, used when validating the launch task commands with `task_gpp` set, see *launch_task()*.

A sample platform configuration file for a workstation. It assumes that the workstation:

- does not have a batch scheduler or resource manager
- may have multiple cores and sockets
- does not have portal access
- will manually specify the allocation

```
HOST = workstation
MPIRUN = mpirun # eval

########################################
# resource detection method
########################################
NODE_DETECTION = manual # checkjob | qstat | pbs_env | slurm_env | manual

########################################
# manual allocation description
########################################
TOTAL_PROCS = 4
NODES = 1
PROCS_PER_NODE = 4

########################################
# node topology description
########################################
CORES_PER_NODE = 4
SOCKETS_PER_NODE = 1

########################################
# framework setting for node allocation
```

(continues on next page)

[3] Only used if manual allocation is specified, or if no detection mechanism is specified and none of the other mechanisms work first. It is the *users* responsibility for this value to make sense.

[2] Used in manual allocation detection and will override any detected ppn value (if smaller than the machine maximum ppn).

[1] This value should not change unless the machine is upgraded to a different architecture or implements different allocation policies.

(continued from previous page)

```
########################################
# MUST ADHERE TO THE PLATFORM'S CAPABILITIES
#   * EXCLUSIVE : only one task per node
#   * SHARED : multiple tasks may share a node
# For single node jobs, this can be overridden allowing multiple
# tasks per node.
NODE_ALLOCATION_MODE = SHARED # SHARED | EXCLUSIVE
```

3.4 Developing Drivers and Components for IPS Simulations

This section is for those who wish to modify and write drivers and components to construct a new simulation scenario. It is expected that readers are familiar with IPS terminology, the directory structure and have looked at some existing drivers and components before attempting to modify or create new ones. This guide will describe the elements of a simulation, how they work together, the structure of drivers and components, IPS services API, and a discussion of control flow, data flow and fault management.

3.4.1 Development environment

It is suggested that for developing drivers and component that you use a separate conda environment to your production environment using the latest stable release of IPS. See *Building and Setting up Your Environment*.

It is also recommended to write components as there own packages, see *Create a component package*.

Elements of a Simulation

When constructing a new simulation scenario, writing a new component or even making small modifications to existing components and drivers, it is important to consider and understand how the pieces of an IPS simulation work together. An IPS simulation scenario is specified in the *configuration file*. This file tells the framework how to set up the output tree for the data files, which components are needed and where the implementation is located, time loop and checkpoint parameters, and input and output files for each component and the simulation as a whole are specified. The *framework* uses this information to find the pieces of code and data that come together to form the simulation, as well as provide this information to the components and driver to manage the simulation and execution of tasks[1].

The framework provides *services* that are used by components to perform data, task, resource and configuration management, and provides an event service for exchanging messages with internal and external entities. While these services are provided as a single API to component writers, the documentation (and underlying implementation) divides them into groups of methods to perform related actions. *Data management* services include staging input, output and plasma state files, changing directories, and saving task restart files, among others. The framework will perform these actions for the calling component based on the files specified in the configuration file and within the method call maintaining coherent directory spaces for each component's work, previous steps, checkpoints and globally accessible data to insure that name collisions do not corrupt data and that global files are accessed in a well-defined manner[2]. Services for *task management* include methods for component method invocations, or *calls*, and executable launch on compute nodes, or *task launches*. The task management portion of the framework works in conjunction with the IPS resource manager to execute multiple parallel executables within a single batch allocation, allowing IPS simulations to efficiently utilize

[1] Tasks are the binaries that are launched by components on compute nodes, where as components are Python scripts that manage the data movements and execution of the tasks (with the help of IPS services). In general, the component is aware of the driver and its existence within a coupled simulation, and the task does not.

[2] The IPS uses an agreed upon file format and associated library to manage global (shared) data for the simulation, called the Plasma State. It is made up of a set of netCDF files with a defined layout so that codes can access and share the data. At the beginning of each step the component will get a local copy of the current plasma state, execute based on these values, and then update the plasma state values that it changed to the global copy. There are data management services to perform these actions, see *Data Management API*.

compute resources, as data dependencies allow. The IPS task manager provides blocking and non-blocking versions of `call` and `launch_task`, including a notion of *task pools* and the ability to wait for the completion of any or all calls or tasks in a group. These different invocation and launch methods allow a component writer to manage the control flow and implement data dependencies between components and tasks. This task management interface hides the resource management, platform specific, task scheduling, and process interactions that are performed by the framework, allowing component writers to express their simulations and component coupling more simply. The *configuration manager* primarily reads the configuration file and instantiates the components for the simulation so that they can interact over the course of the simulation. It also provides an interface for accessing key data elements from the configuration file, such as the time loop, handles to components and any component specific items listed in the configuration file.

Components

There are three classes of components: framework, driver, and general purpose (physics components fall into this category). In the IPS, each component executes in a separate process (a child of the framework) and implements the following methods:

`init(self, timeStamp=0)` This function performs pre-simulation setup activities such as reading in global configuration parameters, checking configuration parameters, updating input files and internal state. (Component configuration parameters are populated *before* `init` is ever called.)

`step(self, timeStamp=0)` This function is the main part of the component. It is responsible for launching any tasks, and managing the input, output and plasma state during the course of the step.

`finalize(self, timeStamp=0)` This function is called after the simulation has completed and performs any clean up that is required by the component. Typically there is nothing to do.

`checkpoint(self, timeStamp=0)` This function performs a checkpoint for the component. All of the files marked as restart files in the configuration file are automatically staged to the checkpoint area. If the component has any internal knowledge or logic, or if there are any additional files that are needed to restart, this should be done explicitly here.

`restart(self, timeStamp=0)` This function replaces `init` when restarting a simulation from a previous simulation step. It should read in data from the appropriate files and set up the component so that it is ready to compute the next step.

The component writer will use the services API to help perform data, task, configuration and event management activities to implement these methods.

This document focuses on helping (physics) component and driver writers successfully write new components. It will take the writer step-by-step through the process of writing basic components.

Writing Components

In this section, we take you through the steps of adding a new component to the IPS landscape. It will cover where to put source code, scripts, binaries and inputs, how to construct the component, how to add the component to the IPS build system, and some tips to make this process smoother.

Adding a New Binary

The location of the binary does not technically matter to the framework, as long as the path can be constructed by the component and the permissions are set properly to launch it when the time comes. There are two recommended ways to express the location of the binary to the component:

1. For stable and shared binaries, the convention is to put them in the platform's *PHYS_BIN*. This way, the *PHYS_BIN* is specified in the platform configuration file and the component can access the location of the binary relative to that location on each machine. See *Platforms and Platform Configuration*.
2. The location of the binary is specified in the component's section of the simulation configuration file. This way, the binary can be specified just before runtime and the component can access it through the framework services. This convention is typically used during testing, experimentation with new features in the code, or other circumstances where the binary may not be stable, fully compatible with other components, or ready to be shared widely.

Data Coupling Preparation

Once you have your binary built properly and available, it is time to work on the data coupling to the other components in a simulation. This is a component specific task, but it often takes conversation with the other physicists in the group as to what values need to be communicated and to develop an understanding of how they are used.

When the physics of interest is identified, adapters need to be written to translate IPS-style inputs (from the Plasma State) to the inputs the binary is expecting, and a similar adapter for the output files.

Create a Component

Now it is time to start writing the component. At this point you should have an idea of how the component will fit into a coupled simulation and the types of activities that will need to happen during the *init*, *step*, and *finalize* phases of execution.

1. Create a directory for your component (if you haven't already). The convention in the IPS repository is to put component scripts and helpers in `ips/components/<port_name>/<component_name>`, where *port_name* is the "type" of component, and the *component_name* is the implementation of that "type" of component. Often, *component_name* will contain the name of the code it executes. If there is already a component directory and existing components, then you may want to make your own directory within the existing component's directory or just add your component in that same directory.
2. Copy the skeleton component (`ips/doc/examples/skeleton_comp.py`) to the directory you choose or created. Be sure to name it such that others will easily know what the component does. For example, a component for TORIC, a code that models radio frequency heating in plasmas, is found in `ips/components/rf/toric/` and called `rf_ic_toric_mcmd.py`.
3. Edit skeleton. Components should be written such that the inputs, outputs, binaries and other parameters are specified in the configuration file or appear in predictable locations *across platforms*. The skeleton contains an outline, in comments, of the activities that a generic component does in each method invocation. You will need to fill in the outline with your own calls to the services and any additional activities in the appropriate places. Take a look at the other example components in the `ips/doc/examples/` or `ips/components/` for guidance. The following is an outline of the changes that need to be made:
    a. Change the name of the class and update the file to use that name every where it says `# CHANGE EXAMPLE TO COMPONENT NAME`.
    b. Modify `init` to initialize the input files that are needed for the first step. Update shared files as needed.

c. Modify `step` to use the appropriate *prepare_input* and *process_output* executables. Make sure all shared files that are changed during the course of the task execution are saved to their proper locations for use by other components. Make sure that all output files that are needed for the next step are copied to archival location. If a different task launch mechanism is required, modify as needed. See *Task Launch API* for related services.

d. Modify `finalize` to do any clean up as needed.

e. Modify `checkpoint` to save all files that are needed to restart from later.

f. Modify `restart` to set up the component to resume computation from a checkpointed step.

While writing your component, be sure to use `try...except` blocks[3] to catch problems and the services logging mechanisms to report critical errors, warnings, info and debug messages. It is *strongly* recommended that you use exceptions and the services logging capability for debugging and output. Not catching exceptions in the component can lead to the driver or framework catching them in a weird place and it will likely take a long time to track down where the problem occurred. The logging mechanism in the IPS provides time stamps of when the event occurred, the component that produced the message, as well as a nice way to format the message information. These messages are written to the log file (specified in the configuration file for the simulation) atomically, unlike normal print statements. Absolute ordering is not guaranteed across different components, but ordering within the same component is guaranteed. See *Logging API* for more information on when to use the different logging levels.

At this point, it might be a good idea to start the documentation of the component in `ips/doc/component_guides/`. You will find a *README* file in `ips/doc/` that explains how to build and write IPS documentation, and another in the `ips/doc/component_guides/` on what information to include in your component documentation.

Testing and Debugging a Component

Now it is time to construct a simulation to test your new component. There are two ways to test a new component. The first is to have the IPS just run that single component without a driver, by specifying your component as the driver. The second is to plug it into an existing driver. The former will test only the task launching and data movement capabilities. The latter can also test the data coupling and call interface to the component. This section will describe how to xstest your component using an existing driver (including how to add the new component to the driver).

As you can see in the example component, almost everything is specified in the configuration file and read at runtime. This means that the configuration of components is vitally important to their success or failure. The entries in the component configuration section are made available to the component automatically, thus a component can access them by *self.<entry_name>*. This is useful in many cases, and you can see in the example component that *self.NPROC* and *self.BIN_PATH* are used. Global configuration parameters can also be accessed using services call *get_config_param(<param_name>)* (*API*).

Drivers access components by their port names (as specified in the configuration file). To add a new component to the driver you will either need to add a new port name or use an existing port name. `ips/components/drivers/dbb/generic_driver.py` is a good all-purpose driver that most components should be able to use. If you are using an existing port name, then the code should just work. It is recommended to go through the driver code to make sure the component is being used in the expected manner. To add a new port name, you will need to add code to *generic_driver.step()*:

- get a reference to the port (*self.services.get_port(<name of port>)*)
- call "init" on that component (*self.services.call(comp_ref, "init")*)
- call "step" on that component (*self.services.call(comp_ref, "step")*)
- call "finalize" on that component (*self.services.call(comp_ref, "finalize")*)

The following sections of the configuration file may need to be modified. If you are not adding the component to an existing simulation, you can copy a configuration file from the examples directory and modify it.

[3] Tutorial on exceptions

1. *Plasma State (Shared Files) Section*

   You will need to modify this section to include any additional files needed by your component:

```
# Where to put plasma state files as the simulation evolves
STATE_WORK_DIR = ${SIM_ROOT}/work/plasma_state
CURRENT_STATE = ${SIM_NAME}_ps.cdf
PRIOR_STATE = ${SIM_NAME}_psp.cdf
NEXT_STATE = ${SIM_NAME}_psn.cdf
CURRENT_EQDSK = ${SIM_NAME}_ps.geq
CURRENT_CQL = ${SIM_NAME}_ps_CQL.nc
CURRENT_DQL = ${SIM_NAME}_ps_DQL.nc
CURRENT_JSDSK = ${RUN_ID}_ps.jso

# What files constitute the plasma state
STATE_FILES1 = ${CURRENT_STATE} ${PRIOR_STATE}
                      ${NEXT_STATE}
STATE_FILES2 = ${STATE_FILES1}  ${CURRENT_EQDSK}
                      ${CURRENT_CQL} ${CURRENT_DQL}
STATE_FILES = ${STATE_FILES2}  ${CURRENT_JSDSK}
```

2. *Ports Section*

   You will need to add the component to the ports section so that it can be properly detected by the framework and driver. An entry for *DRIVER* must be specified, otherwise the framework will abort. Also, a warning is produced if there is no *INIT* component. Note that all components added to the *NAMES* field must have a corresponding subsection.

```
[PORTS]
    NAMES = INIT DRIVER MONITOR EPA NB
   [[DRIVER]]
        IMPLEMENTATION = EPA_IC_FP_NB_DRIVER
    [[INIT]]
        IMPLEMENTATION = minimal_state_init
    [[RF_IC]]
        IMPLEMENTATION = model_RF_IC


    ...
```

3. *Component Description Section*

   The ports section just defines which components are going to be used in this simulation, and point to the section where they are described. The component description section is where those definitions take place:

```
[TSC]
    CLASS = epa
    SUB_CLASS =
    NAME = tsc
    NPROC = 1
    BIN_PATH = /path/to/bin
    INPUT_DIR = /path/to/components/epa/tsc
    INPUT_FILES = inputa.I09001 sprsina.I09001config_nbi_ITER.dat
    OUTPUT_FILES = outputa tsc.cgm inputa log.tsc ${STATE_FILES}
    SCRIPT = ${BIN_PATH}/epa_nb_iter.py
```

   The component section starts with a label that matches what is listed as the implementation in the ports section.

These *must* match or else the framework will not find your component and the simulation will fail before it starts (or worse, use the wrong implementation!). *CLASS* and *SUBCLASS* typically refer to the directory hierarchy and are sometimes used to identify the location of the source code and input files. Note that *NAME* must match the python class name that implements the component. *NPROC* is the number of *processes* that the binary needs to use when launched on compute nodes. If you have pre-built binaries that exist in another location, an additional entry in the component description section may be a convenient place to put it. *INPUT_DIR*, *INPUT_FILES* and *OUTPUT_FILES* specify the location and names of the input and output files, respectively. If a subset of plasma states files is all that is required by the component, they are specified here (*STATE_FILES*). If the entry is omitted, *all* of the plasma state files are used. This prevents the full set of files to be copied to and from the component's work directory on every step, saving time and space. Lastly, *SCRIPT* is the Python script that contains the component code, specifically the Python class in *NAME*. Additionally, any component specific values maybe specified here to control logic or set data values that change often.

4. *Time Loop Section*

   This may need to be modified for your component or the driver that uses your new component. During testing, a small number of steps is appropriate.

```
# Time loop specification (two modes for now) EXPLICIT | REGULAR
# For MODE = REGULAR, the framework uses the variables START, FINISH, and NSTEP
# For MODE = EXPLICIT, the framework uses the variable VALUES (space separated
# list of time values)
[TIME_LOOP]
    MODE = EXPLICIT
    VALUES = 75.000 75.025 75.050 75.075 75.100 75.125
```

Tips

This section contains some useful tips on testing, debugging and documenting your new component.

- General:
  - Naming is important. You do not want the name of your component to overlap with another, so make sure it is unique.
  - Be sure to commit all the files and directories that are needed to build and run your component. This means the executables, Makefiles, component script, helper scripts and input files.
- Testing:
  - To test a new component, first run it as the driver component of a simulation all by itself. This will make sure that the component itself works with the framework.
  - The next step is to have a driver call just your new component to make sure it can be discovered and called by the driver properly.
  - The next step is to determine if the component can exchange global data with another component. To do this run two components in a driver and verify they are exchanging data properly.
  - When testing IPS components and simulations, it may be useful to turn on debugging information in the IPS and the underlying executables.
  - If this is a time stepping simulation, a small number of steps is useful because it will lead to shorter running times, allowing you to submit the job to a debug or other faster turnaround queue.
- Debugging:
  - Add logging messages (*services.info()*, *services.warning()*, etc.) to make sure your component does what you think it does.

  - Remove other components from the simulation to figure out which one or which interaction is causing the problem
  - Take many checkpoints around the problem to narrow in on the problem.
  - Remove concurrency to see if one component is overwriting another's data.
- Documentation:
  - Document the component code such that another person can understand how it works. It helps if the structure remains the same as the example component.
  - Write a description of what the component does, the inputs it uses, outputs it produces, and what scenarios and modes it can be used in in the component documentation section.
- Protected attributes:
  - The following Component attributes are used internally within IPS and are protected so you can not assigned to them:
    * `component_id`
    * `services`
    * `config`
    * `start_time`
    * `method_name`
    * `args`

Writing Drivers

The driver of the simulation manages the control flow and synchronization across components via time stepping or implicit means, thus orchestrating the simulation. There is only one driver per simulation and it is invoked by the framework and is responsible for invoking the components that make up the simulation scenario it implements. It is also responsible for managing data at the simulation level, including checkpoint and restart activities.

Before writing a driver, it is a good idea to have the components already written. Once the components that are to be used are chosen the data coupling and control flow must be addressed.

In order to couple components, the data that must be exchanged between them and the ordering of updates to the plasma state must be determined. Once the data dependencies are identified (which components have to run before the next, and which ones can run at the same time). You can write the body of the driver. Before going through the steps of writing a driver, review the *method invocation API* and plan which methods to use during the main time loop.

The framework will invoke the methods of the *INIT* and *DRIVER* components over the course of the simulation, defining the execution of the run:

- `init_comp.init()` - initialization of initialization component
- `init_comp.step()` - execution of initialization work
- `init_comp.finalize()` - cleanup and confirmation of initialization
- `driver.init()` - any initialization work (typically empty)
- `driver.step()` - the bulk of the simulation
  - get references to the ports
  - call *init* on each port
  - get the time loop

  - implement logic of time stepping
  - during each time step:
    * perform pre-step logic that may stage data or determine which components need to run or what parameters are given to each component
    * call *step* on each port (as appropriate)
    * manage global plasma state at the end of each step
    * checkpoint components (frequency of checkpoints is controlled by framework)
  - call *finalize* on each component
- `driver.finalize()` - any clean up activities (typically empty)

It is recommended that you start with the `ips/components/drivers/dbb/generic_driver.py` and modify it as needed. You will most likely be changing: how the components are called in the main loop (the generic driver calls each component in sequence), the pre-step logic phase, and what ports are used. The data management and checkpointing calls should remain unchanged as their behavior is controlled in the configuration file.

The process for adding a new driver to the IPS is the same as that for the component. See the appropriate sections above for adding a component.

IPS Services API

The IPS framework contains a set of managers that perform services for the components. A component uses the services API to access them, thus hiding the complexity of the framework implementation. Below are descriptions of the individual function calls grouped by type. To call any of these functions in a component replace *ServicesProxy* with *self.services*. The *services* object is passed to the component upon creation by the framework.

Component Invocation

Component invocation in the IPS means one component is calling another component's function. This API provides a mechanism to invoke methods on components through the framework. There are blocking and non-blocking versions, where the non-blocking versions require a second function to check the status of the call. Note that the *wait_call* has an optional argument (*block*) that changes when and what it returns.

`ServicesProxy.call`(*component_id*, *method_name*, **args*, ***keywords*)

Invoke method *method_name* on component *component_id* with optional arguments **args*. Will wait until call is finished. Return result from invoking the method.

Parameters

- **component_id** (*ComponentID*) – Component ID of requested component
- **method_name** (*str*) – component method to call, e.g. `init` or `step`

Returns service response message arguments

`ServicesProxy.call_nonblocking`(*component_id*, *method_name*, **args*, ***keywords*)

Invoke method *method_name* on component *component_id* with optional arguments **args*. Will not wait until finished.

Parameters

- **component_id** (*ComponentID*) – Component ID of requested component
- **method_name** (*str*) – component method to call, e.g. `init` or `step`

Returns call_id

Return type int

ServicesProxy.**wait_call**(*call_id*, *block=True*)

If *block* is True, return when the call has completed with the return code from the call. If *block* is False, raise *IncompleteCallException* if the call has not completed, and the return value is it has.

Parameters **call_id** (*int*) – call ID

Returns service response message arguments

ServicesProxy.**wait_call_list**(*call_id_list*, *block=True*)

Check the status of each of the call in *call_id_list*. If *block* is True, return when *all* calls are finished. If *block* is False, raise *IncompleteCallException* if *any* of the calls have not completed, otherwise return. The return value is a dictionary of *call_ids* and return values.

Parameters **call_id_list** (*list of int*) – list of call ID's

Returns dict of call_id and return value

Return type dict

Task Launch

The task launch interface allows components to launch and manage the execution of (parallel) executables. Similar to the component invocation interface, the behavior of *launch_task()* and the *wait_task()* variants are controlled using the *block* keyword argument and different interfaces to *wait_task*.

The task_ppn and task_cpp options all greater control over how commands are made. task_ppn will limit the number of task per node, task_ccp will limit the number of cores assigned to each process, this is only used when MPIRUN=srun, if task_cpp is not set it will be calculated automatically.

Slurm examples

The following examples show the behavior if you are running on a Cori with 32 cores per node.

Using the check-mpi.gnu.cori binary provided on Cori with nproc=8 and settings the correct OMP environment variables with omp=True without specifying other options :

```
self.services.launch_task(8, cwd, "check-mpi.gnu.cori", omp=True)
```

the srun command created will be srun -N 1 -n 8 -c 4 --threads-per-core=1 --cpu-bind=cores check-mpi.gnu.cori along with settings the environment variables for OpenMP OMP_PLACES=threads OMP_PROC_BIND=spread OMP_NUM_THREADS=4. The resulting core affinity is

```
Hello from rank 0, on nid00025. (core affinity = 0-3)
Hello from rank 1, on nid00025. (core affinity = 16-19)
Hello from rank 2, on nid00025. (core affinity = 4-7)
Hello from rank 3, on nid00025. (core affinity = 20-23)
Hello from rank 4, on nid00025. (core affinity = 8-11)
Hello from rank 5, on nid00025. (core affinity = 24-27)
Hello from rank 6, on nid00025. (core affinity = 12-15)
Hello from rank 7, on nid00025. (core affinity = 28-31)
```

If you also include the option task_ppn=4:

```
self.services.launch_task(8, cwd, "check-mpi.gnu.cori", task_ppn=4, omp=True)
```

then the command created will be `srun -N 2 -n 8 -c 8 --threads-per-core=1 --cpu-bind=cores check-mpi.gnu.cori` along with settings the environment variables for OpenMP `OMP_PLACES=threads OMP_PROC_BIND=spread OMP_NUM_THREADS=8`. The resulting core affinity is

```
Hello from rank 0, on nid00025. (core affinity = 0-7)
Hello from rank 1, on nid00025. (core affinity = 16-23)
Hello from rank 2, on nid00025. (core affinity = 8-15)
Hello from rank 3, on nid00025. (core affinity = 24-31)
Hello from rank 4, on nid00026. (core affinity = 0-7)
Hello from rank 5, on nid00026. (core affinity = 16-23)
Hello from rank 6, on nid00026. (core affinity = 8-15)
Hello from rank 7, on nid00026. (core affinity = 24-31)
```

You can limit the `--cpus-per-task` of the `srun` command by setting `task_cpp`, adding `task_cpp=2`

```
self.services.launch_task(8, cwd, "check-mpi.gnu.cori", task_ppn=4, task_cpp=2, omp=True)
```

will create the command `srun -N 2 -n 8 -c 2 --threads-per-core=1 --cpu-bind=cores check-mpi.gnu.cori` and set `OMP_PLACES=threads OMP_PROC_BIND=spread OMP_NUM_THREADS=2`. This will result in under-utilizing the nodes, which may be needed if your task is memory bound. The resulting core affinity is

```
Hello from rank 0, on nid00025. (core affinity = 0,1)
Hello from rank 1, on nid00025. (core affinity = 16,17)
Hello from rank 2, on nid00025. (core affinity = 2,3)
Hello from rank 3, on nid00025. (core affinity = 18,19)
Hello from rank 4, on nid00026. (core affinity = 0,1)
Hello from rank 5, on nid00026. (core affinity = 16,17)
Hello from rank 6, on nid00026. (core affinity = 2,3)
Hello from rank 7, on nid00026. (core affinity = 18,19)
```

Using the check-hybrid.gnu.cori binary with the same options:

```
self.services.launch_task(8, cwd, "check-hybrid.gnu.cori", task_ppn=4, task_cpp=2,␣
↪omp=True)
```

the resulting core affinity of the OpenMP threads are:

```
Hello from rank 0, thread 0, on nid00025. (core affinity = 0)
Hello from rank 0, thread 1, on nid00025. (core affinity = 1)
Hello from rank 1, thread 0, on nid00025. (core affinity = 16)
Hello from rank 1, thread 1, on nid00025. (core affinity = 17)
Hello from rank 2, thread 0, on nid00025. (core affinity = 2)
Hello from rank 2, thread 1, on nid00025. (core affinity = 3)
Hello from rank 3, thread 0, on nid00025. (core affinity = 18)
Hello from rank 3, thread 1, on nid00025. (core affinity = 19)
Hello from rank 4, thread 0, on nid00026. (core affinity = 0)
Hello from rank 4, thread 1, on nid00026. (core affinity = 1)
Hello from rank 5, thread 0, on nid00026. (core affinity = 16)
Hello from rank 5, thread 1, on nid00026. (core affinity = 17)
Hello from rank 6, thread 0, on nid00026. (core affinity = 2)
Hello from rank 6, thread 1, on nid00026. (core affinity = 3)
Hello from rank 7, thread 0, on nid00026. (core affinity = 18)
Hello from rank 7, thread 1, on nid00026. (core affinity = 19)
```

Slurm with GPUs examples

Note: New in 0.8.0

The *launch_task()* method has an option `task_gpp` which allows you to set the number of GPUs per process, used as the `--gpus-per-task` in the `srun` command.

IPS will validate the number of GPUs per node requested does not exceed the number specified by the `GPUS_PER_NODE` parameter in the *Platform Configuration File*. You need to make sure that the number of GPUs per process times the number of processes per node does not exceed the `GPUS_PER_NODE` set.

Using the gpus_for_tasks program provided for Perlmutter (which has 4 GPUs per node) to test the behavior, you will see the following:

To launch a task with 1 process and 1 GPU per process (`task_gpp`) run:

```
self.services.launch_task(1, cwd, "gpu-per-task", task_gpp=1)
```

will create the command `srun -N 1 -n 1 -c 64 --threads-per-core=1 --cpu-bind=cores --gpus-per-task=1 gpus_for_tasks` and the output of will be:

```
Rank 0 out of 1 processes: I see 1 GPU(s).
0 for rank 0: 0000:03:00.0
```

To launch 8 processes on 2 nodes (so 4 processes per node) with 1 gpu per process run:

```
self.services.launch_task(8, cwd, "gpu-per-task", task_ppn=4, task_gpp=1)
```

will create the command `srun -N 2 -n 8 -c 16 --threads-per-core=1 --cpu-bind=cores --gpus-per-task=1 gpus_for_task` and the output of will be:

```
Rank 0 out of 8 processes: I see 1 GPU(s).
0 for rank 0: 0000:03:00.0
Rank 1 out of 8 processes: I see 1 GPU(s).
0 for rank 1: 0000:41:00.0
Rank 2 out of 8 processes: I see 1 GPU(s).
0 for rank 2: 0000:82:00.0
Rank 3 out of 8 processes: I see 1 GPU(s).
0 for rank 3: 0000:C1:00.0
Rank 4 out of 8 processes: I see 1 GPU(s).
0 for rank 4: 0000:03:00.0
Rank 5 out of 8 processes: I see 1 GPU(s).
0 for rank 5: 0000:41:00.0
Rank 6 out of 8 processes: I see 1 GPU(s).
0 for rank 6: 0000:82:00.0
Rank 7 out of 8 processes: I see 1 GPU(s).
0 for rank 7: 0000:C1:00.0
```

To launch 2 processes on 2 nodes (so 1 processes per node) with 4 gpu per process run:

```
self.services.launch_task(2, cwd, "gpu-per-task", task_ppn=1, task_gpp=4)
```

will create the command `srun -N 2 -n 2 -c 64 --threads-per-core=1 --cpu-bind=cores --gpus-per-task=4 gpus_per_tasks` and the output of will be:

```
Rank 0 out of 2 processes: I see 4 GPU(s).
0 for rank 0: 0000:03:00.0
1 for rank 0: 0000:41:00.0
2 for rank 0: 0000:82:00.0
3 for rank 0: 0000:C1:00.0
Rank 1 out of 2 processes: I see 4 GPU(s).
0 for rank 1: 0000:03:00.0
1 for rank 1: 0000:41:00.0
2 for rank 1: 0000:82:00.0
3 for rank 1: 0000:C1:00.0
```

If you try to launch a task with too many GPUs per node, *e.g.*:

```
self.services.launch_task(8, cwd, "gpu-per-task", task_gpp=1)
```

then it will raise an *GPUResourceRequestMismatchException*.

ServicesProxy.**launch_task**(*nproc*, *working_dir*, *binary*, **args*, ***keywords*)

Launch *binary* in *working_dir* on *nproc* processes. **args* are any arguments to be passed to the binary on the command line. ***keywords* are any keyword arguments used by the framework to manage how the binary is launched. Keywords may be the following:

- *task_ppn* : the processes per node value for this task
- *task_cpp* : the cores per process, only used when `MPIRUN=srun` commands
- *task_gpp* : the gpus per process, only used when `MPIRUN=srun` commands
- ***omp*** [If `True` the task will be launch with the correct OpenMP environment] variables set, only used when `MPIRUN=srun`
- *block* : specifies that this task will block (or raise an exception) if not enough resources are available to run immediately. If `True`, the task will be retried until it runs. If `False`, an exception is raised indicating that there are not enough resources, but it is possible to eventually run. (default = `True`)
- *tag* : identifier for the portal. May be used to group related tasks.
- *logfile* : file name for `stdout` (and `stderr`) to be redirected to for this task. By default `stderr` is redirected to `stdout`, and `stdout` is not redirected.
- *whole_nodes* : if `True`, the task will be given exclusive access to any nodes it is assigned. If `False`, the task may be assigned nodes that other tasks are using or may use.
- *whole_sockets* : if `True`, the task will be given exclusive access to any sockets of nodes it is assigned. If `False`, the task may be assigned sockets that other tasks are using or may use.
- *launch_cmd_extra_args* : extra command arguments added the the MPIRUN command

Return *task_id* if successful. May raise exceptions related to opening the logfile, being unable to obtain enough resources to launch the task (*InsufficientResourcesException*), bad task launch request (*ResourceRequestMismatchException*, *BadResourceRequestException*) or problems executing the command. These exceptions may be used to retry launching the task as appropriate.

Note: This is a nonblocking function, users must use a version of *ServicesProxy.wait_task()* to get result.

Parameters

- **nproc** (*int*) – number of processes

- **working_dir** (*str*) – change to this directory before launching task
- **binary** (*str*) – command to execute, can include arguments or can be pass in with **args*

Returns task_id (PID)

Return type int

ServicesProxy.**wait_task**(*task_id*, *timeout=- 1*, *delay=1*)
Check the status of task *task_id*. Return the return value of the task when finished successfully. Raise exceptions if the task is not found, or if there are problems finalizing the task.

Parameters

- **task_id** (*int*) – task ID (PID)
- **timeout** (*float*) – maximum time to wait for task to finish, default -1 (no timeout)
- **delay** (*float*) – time to wait before checking if task has timed-out

Returns return value of task

ServicesProxy.**wait_task_nonblocking**(*task_id*)
Check the status of task *task_id*. If it has finished, the return value is populated with the actual value, otherwise `None` is returned. A *KeyError* exception may be raised if the task is not found.

Parameters **task_id** (*int*) – task ID (PID)

Returns return value of task if finished else None

ServicesProxy.**wait_tasklist**(*task_id_list*, *block=True*)
Check the status of a list of tasks. If `block` is `True`, return a dictionary of return values when *all* tasks have completed. If `block` is `False`, return a dictionary containing entries for each *completed* task. Note that the dictionary may be empty. Raise `KeyError` exception if `task_id` not found.

Parameters

- **task_id_list** (*list of int*) – list of task_id's (PID's) to wait until completed
- **block** (*bool*) – if to wait until all task finish

Returns dict of task_id and return value

Return type dict

ServicesProxy.**kill_task**(*task_id*)
Kill launched task *task_id*. Return if successful. Raises exceptions if the task or process cannot be found or killed successfully.

Parameters **task_id** (*int*) – task ID

Returns if successfully killed

Return type bool

ServicesProxy.**kill_all_tasks**()
Kill all tasks associated with this component.

The task pool interface is designed for running a group of tasks that are independent of each other and can run concurrently. The services manage the execution of the tasks efficiently for the component. Users must first create an empty task pool, then add tasks to it. The tasks are submitted as a group and checked on as a group. This interface is basically a wrapper around the interface above for convenience.

ServicesProxy.**create_task_pool**(*task_pool_name*)
Create an empty pool of tasks with the name *task_pool_name*. Raise exception if duplicate name.

ServicesProxy.**add_task**(*task_pool_name*, *task_name*, *nproc*, *working_dir*, *binary*, **args*, ***keywords*)
Add task *task_name* to task pool *task_pool_name*. Remaining arguments are the same as in *ServicesProxy.launch_task()*.

ServicesProxy.**submit_tasks**(*task_pool_name*, *block=True*, *use_dask=False*, *dask_nodes=1*, *dask_ppw=None*, *launch_interval=0.0*, *use_shifter=False*, *shifter_args=None*, *dask_worker_plugin=None*, *dask_worker_per_gpu=False*)
Launch all unfinished tasks in task pool *task_pool_name*. If *block* is `True`, return when all tasks have been launched. If *block* is `False`, return when all tasks that can be launched immediately have been launched. Return number of tasks submitted.

Optionally, dask can be used to schedule and run the task pool.

ServicesProxy.**get_finished_tasks**(*task_pool_name*)
Return dictionary of finished tasks and return values in task pool *task_pool_name*. Raise exception if no active or finished tasks.

ServicesProxy.**remove_task_pool**(*task_pool_name*)
Kill all running tasks, clean up all finished tasks, and delete task pool.

Miscellaneous

The following services do not fit neatly into any of the other categories, but are important to the execution of the simulation.

ServicesProxy.**get_working_dir**()
Return the working directory of the calling component.

The structure of the working directory is defined using the configuration parameters *CLASS*, *SUB_CLASS*, and *NAME* of the component configuration section. The structure of the working directory is:

```
${SIM_ROOT}/work/$CLASS_${SUB_CLASS}_$NAME_<instance_num>
```

Returns working directory

Return type str

ServicesProxy.**update_time_stamp**(*new_time_stamp=- 1*)
Update time stamp on portal.

ServicesProxy.**send_portal_event**(*event_type='COMPONENT_EVENT'*, *event_comment=''*, *event_time=None*, *elapsed_time=None*)
Send event to web portal.

Data Management

The data management services are used by the components to manage the data needed and produced by each step, and for the driver to manage the overall simulation data. There are methods for component local, and simulation global files. Fault tolerance services are presented in another section.

Staging of local (non-shared) files:

ServicesProxy.**stage_input_files**(*input_file_list*)
Copy component input files to the component working directory (as obtained via a call to *ServicesProxy.get_working_dir()*). Input files are assumed to be originally located in the directory variable *INPUT_DIR* in the component configuration section.

File are copied using *ipsframework.ipsutil.copyFiles()*.

Parameters **input_file_list** (*str or Iterable of str*) – input files can space separated string or iterable of strings

ServicesProxy.**stage_output_files**(*timeStamp*, *file_list*, *keep_old_files=True*, *save_plasma_state=True*)
Copy associated component output files (from the working directory) to the component simulation results directory. Output files are prefixed with the configuration parameter *OUTPUT_PREFIX*. The simulation results directory has the format:

```
${SIM_ROOT}/simulation_results/<timeStamp>/components/$CLASS_${SUB_CLASS}_$NAME_$
↪{SEQ_NUM}
```

Additionally, plasma state files are archived for debugging purposes:

```
${SIM_ROOT}/history/plasma_state/<file_name>_$CLASS_${SUB_CLASS}_$NAME_<timeStamp>
```

Copying errors are not fatal (exception raised).

Staging of global (plasma state) files:

ServicesProxy.**stage_state**(*state_files=None*)
Copy current state to work directory.

ServicesProxy.**update_state**(*state_files=None*)
Copy local (updated) state to global state. If no state files are specified, component configuration specification is used. Raise exceptions upon copy.

ServicesProxy.**merge_current_state**(*partial_state_file*, *logfile=None*, *merge_binary=None*)
Merge partial plasma state with global state. Partial plasma state contains only the values that the component contributes to the simulation. Raise exceptions on bad merge. Optional *logfile* will capture `stdout` from merge. Optional *merge_binary* specifies path to executable code to do the merge (default value : "update_state")

Configuration Parameter Access

These methods access information from the simulation configuration file.

ServicesProxy.**get_port**(*port_name*)

Parameters **port_name** (*str*) – port name

Returns Return a reference to the component implementing port *port_name*.

Return type *ipsframework.componentRegistry.ComponentID*

ServicesProxy.**get_config_param**(*param*, *silent=False*)
Return the value of the configuration parameter `param`. Raise exception if not found and silent is False.

Parameters

- **param** (*str*) – The parameter requested from simulation config
- **silent** (*bool*) – If True and parameter isn't found then exception is not raised, default False

Returns dictionary of given parameter from configuration

Return type dict

ServicesProxy.**set_config_param**(*param*, *value*, *target_sim_name=None*)
Set configuration parameter *param* to *value*. Raise exceptions if the parameter cannot be changed or if there are problems setting the value. This tell the framework to call *ipsframework.configurationManager.ConfigurationManager.set_config_parameter()* to change the parameter.

Parameters

- **param** (*str*) – The parameter requested from simulation config
- **value** – The value to set the parameter

Returns return value from setting parameter

ServicesProxy.**get_time_loop**()

Return the list of times as specified in the configuration file.

Returns list of times

Return type list of float

Logging

The following logging methods can be used to write logging messages to the simulation log file. It is *strongly* recommended that these methods are used as opposed to print statements. The logging capability adds a timestamp and identifies the component that generated the message. The syntax for logging is a simple string or formatted string:

```
self.services.info('beginning step')
self.services.warning('unable to open log file %s for task %d, will use stdout instead',
                      logfile, task_id)
```

There is no need to include information about the component in the message as the IPS logging interface includes a time stamp and information about what component sent the message:

```
2011-06-13 14:17:48,118 drivers_ssfoley_branch_test_driver_1 DEBUG    __initialize__():
↪<branch_testing.branch_test_driver object at 0xb600d0>  branch_testing_hopper@branch_
↪test_driver@1
2011-06-13 14:17:48,125 drivers_ssfoley_branch_test_driver_1 DEBUG    Working directory /
↪scratch/scratchdirs/ssfoley/rm_dev/branch_testing_hopper/work/drivers_ssfoley_branch_
↪test_driver_1 does not exist - will attempt creation
2011-06-13 14:17:48,129 drivers_ssfoley_branch_test_driver_1 DEBUG    Running - CompID =␣
↪ branch_testing_hopper@branch_test_driver@1
2011-06-13 14:17:48,130 drivers_ssfoley_branch_test_driver_1 DEBUG    _init_event_
↪service(): self.counter = 0 - <branch_testing.branch_test_driver object at 0xb600d0>
2011-06-13 14:17:51,934 drivers_ssfoley_branch_test_driver_1 INFO     ('Received Message
↪',)
2011-06-13 14:17:51,934 drivers_ssfoley_branch_test_driver_1 DEBUG    Calling method␣
↪init args = (0,)
2011-06-13 14:17:51,938 drivers_ssfoley_branch_test_driver_1 INFO     ('Received Message
↪',)
2011-06-13 14:17:51,938 drivers_ssfoley_branch_test_driver_1 DEBUG    Calling method␣
↪step args = (0,)
2011-06-13 14:17:51,939 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪init_task  (48, 'hw', 0, True, True, True)
2011-06-13 14:17:51,939 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|0)
2011-06-13 14:17:51,952 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|0),␣
↪response = <messages.ServiceResponseMessage object at 0xb60ad0>
2011-06-13 14:17:51,954 drivers_ssfoley_branch_test_driver_1 DEBUG    Launching command␣
↪: aprun -n 48 -N 24 -L 1087,1084 hw
2011-06-13 14:17:51,961 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪getTopic  ('_IPS_MONITOR',)
```

(continues on next page)

(continued from previous page)

```
2011-06-13 14:17:51,962 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|1)
2011-06-13 14:17:51,972 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|1),␣
↪response = <messages.ServiceResponseMessage object at 0xb60b90>
2011-06-13 14:17:51,972 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪sendEvent  ('_IPS_MONITOR', 'PORTAL_EVENT', {'sim_name': 'branch_testing_hopper',
↪'portal_data': {'comment': 'task_id = 1 , Tag = None , Target = aprun -n 48 -N 24 -L␣
↪1087,1084 hw ', 'code': 'drivers_ssfoley_branch_test_driver', 'ok': 'True', 'eventtype
↪': 'IPS_LAUNCH_TASK', 'state': 'Running', 'walltime': '4.72'}})
2011-06-13 14:17:51,973 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|2)
2011-06-13 14:17:51,984 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|2),␣
↪response = <messages.ServiceResponseMessage object at 0xb60d10>
2011-06-13 14:17:51,987 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪getTopic  ('_IPS_MONITOR',)
2011-06-13 14:17:51,988 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|3)
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|3),␣
↪response = <messages.ServiceResponseMessage object at 0xb60890>
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪sendEvent  ('_IPS_MONITOR', 'PORTAL_EVENT', {'sim_name': 'branch_testing_hopper',
↪'portal_data': {'comment': 'task_id = 1  elapsed time = 0.00 S', 'code': 'drivers_
↪ssfoley_branch_test_driver', 'ok': 'True', 'eventtype': 'IPS_TASK_END', 'state':
↪'Running', 'walltime': '4.75'}})
2011-06-13 14:17:52,000 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|4)
2011-06-13 14:17:52,012 drivers_ssfoley_branch_test_driver_1 DEBUG    _get_service_
↪response(REQUEST|branch_testing_hopper@branch_test_driver@1|FRAMEWORK@Framework@0|4),␣
↪response = <messages.ServiceResponseMessage object at 0xb60a90>
2011-06-13 14:17:52,012 drivers_ssfoley_branch_test_driver_1 DEBUG    _invoke_service():␣
↪finish_task  (1L, 1)
```

The table below describes the levels of logging available and when to use each one. These levels are also used to determine what messages are produced in the log file. The default level is `WARNING`, thus you will see `WARNING`, `ERROR` and `CRITICAL` messages in the log file.

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. "disk space low"). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

For more information about the logging module and how to used it, see Logging Tutorial.

ServicesProxy.**log**(*msg*, **args*)
Wrapper for *ServicesProxy.info()*.

ServicesProxy.**debug**(*msg*, **args*)
Produce **debugging** message in simulation log file. See `logging.debug()` for usage.

ServicesProxy.**info**(*msg*, **args*)
Produce **informational** message in simulation log file. See `logging.info()` for usage.

ServicesProxy.**warning**(*msg*, **args*)
Produce **warning** message in simulation log file. See `logging.warning()` for usage.

ServicesProxy.**error**(*msg*, **args*)
Produce **error** message in simulation log file. See `logging.error()` for usage.

ServicesProxy.**exception**(*msg*, **args*)
Produce **exception** message in simulation log file. See `logging.exception()` for usage.

ServicesProxy.**critical**(*msg*, **args*)
Produce **critical** message in simulation log file. See `logging.critical()` for usage.

Fault Tolerance

The IPS provides services to checkpoint and restart a coupled simulation by calling the checkpoint and restart methods of each component and certain settings in the configuration file. The driver can call *checkpoint_components*, which will invoke the checkpoint method on each component associated with the simulation. The component's *checkpoint* method uses *save_restart_files* to save files needed by the component to restart from the same point in the simulation. When the simulation is in restart mode, the *restart* method of the component is called to initialize the component, instead of the *init* method. The *restart* component method uses the *get_restart_files* method to stage in inputs for continuing the simulation.

ServicesProxy.**save_restart_files**(*timeStamp*, *file_list*)
Copy files needed for component restart to the restart directory:

```
${SIM_ROOT}/restart/$timestamp/components/$CLASS_${SUB_CLASS}_$NAME
```

Copying errors are not fatal (exception raised).

ServicesProxy.**checkpoint_components**(*comp_id_list*, *time_stamp*, *Force=False*, *Protect=False*)
Selectively checkpoint components in *comp_id_list* based on the configuration section *CHECKPOINT*. If *Force* is `True`, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is `True`, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to `False`.

The *CHECKPOINT_MODE* option controls determines if the components checkpoint methods are invoked.

Possible *MODE* options are:

ALL: Checkpint every time the call is made (equivalent to always setting Force =True)

WALLTIME_REGULAR: checkpoints are saved upon invocation of the service call `checkpoint_components()`, when a time interval greater than, or equal to, the value of the configuration parameter WALLTIME_INTERVAL had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to `checkpoint_components()` before WALLTIME_INTERVAL seconds have passed since the last successful checkpoint result in a NOOP.

WALLTIME_EXPLICIT: checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable WALLTIME_VALUES. Let [t_0, t_1, ..., t_n] be the list of wall clock time values specified in the configuration parameter WALLTIME_VALUES.

Then checkpoint(T) = True if T >= t_j, for some j in [0,n] and there is no other time T_1, with T > T_1 >= T_j such that checkpoint(T_1) = True. If the test fails, the call results in a NOOP.

PHYSTIME_REGULAR: checkpoints are saved at regularly spaced "physics time" intervals, specified in the configuration parameter PHYSTIME_INTERVAL. Let PHYSTIME_INTERVAL = PTI, and the physics time stamp argument in the call to checkpoint_components() be pts_i, with i = 0, 1, 2, … Then checkpoint(pts_i) = True if pts_i >= n PTI , for some n in 1, 2, 3, … and pts_i - pts_prev >= PTI, where checkpoint(pts_prev) = True and pts_prev = max (pts_0, pts_1, ..pts_i-1). If the test fails, the call results in a NOOP.

PHYSTIME_EXPLICIT: checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable PHYSTIME_VALUES. Let [pt_0, pt_1, …, pt_n] be the list of physics time values specified in the configuration parameter PHYSTIME_VALUES. Then checkpoint(pt) = True if pt >= pt_j, for some j in [0,n] and there is no other physics time pt_k, with pt > pt_k >= pt_j such that checkpoint(pt_k) = True. If the test fails, the call results in a NOOP.

The configuration parameter NUM_CHECKPOINT controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of NUM_CHECKPOINT are:

- NUM_CHECKPOINT = n, with n > 0 –> Keep the most recent n checkpoints
- NUM_CHECKPOINT = 0 –> No checkpoints are made/kept (except when *Force* = `True`)
- NUM_CHECKPOINT < 0 –> Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

`ServicesProxy.`**`get_restart_files`**`(`*restart_root*, *timeStamp*, *file_list*`)`
Copy files needed for component restart from the restart directory:

```
<restart_root>/restart/<timeStamp>/components/$CLASS_${SUB_CLASS}_$NAME_${SEQ_NUM}
```

to the component's work directory.

Copying errors are not fatal (exception raised).

Event Service

The event service interface is used to implement the web portal connection, as well as for components to communicate asynchronously.

`ServicesProxy.`**`publish`**`(`*topicName*, *eventName*, *eventBody*`)`
Publish event consisting of *eventName* and *eventBody* to topic *topicName* to the IPS event service.

`ServicesProxy.`**`subscribe`**`(`*topicName*, *callback*`)`
Subscribe to topic *topicName* on the IPS event service and register *callback* as the method to be invoked when an event is published to that topic.

`ServicesProxy.`**`unsubscribe`**`(`*topicName*`)`
Remove subscription to topic *topicName*.

`ServicesProxy.`**`process_events`**`()`
Poll for events on subscribed topics.

3.5 Create a component package

This is an example creating a hello world component installable package. This is also an example of using `MODULE` instead of `SCRIPT` in the component configuration section.

The examples will be a simple hello world with one driver and one worker. The only requirement of the package is `ipsframework`. The ipsframework should be automatically installed from pypi when install ipsexamples but you can manually install it from pypi with

```
python -m pip install ipsframework
```

Or to install it directly from github you can do

```
python -m pip install git+https://github.com/HPC-SimTools/IPS-framework.git
```

To create this project locally, create the following file structure

```
helloworld
├── helloworld
│   ├── __init__.py
│   ├── hello_driver.py
│   └── hello_worker.py
└── setup.py
```

The file `__init__.py` is just empty but turns the *helloworld* folder into a python module.

A simple `setup.py` would be

```
#!/usr/bin/env python3
from setuptools import setup, find_packages

setup(
    name="helloworld",
    version="1.0.0",
    install_requires=["ipsframework"],
    packages=find_packages(),
)
```

The `hello_driver.py` in the most simplest form would be

```
from ipsframework import Component


class hello_driver(Component):
    def __init__(self, services, config):
        super().__init__(services, config)
        print('Created %s' % (self.__class__))

    def step(self, timestamp=0.0):
        print('hello_driver: beginning step call')
        worker_comp = self.services.get_port('WORKER')
        self.services.call(worker_comp, 'step', 0.0)
        print('hello_driver: finished step call')
```

And the `hello_worker.py` is

```
from ipsframework import Component


class hello_worker(Component):
    def __init__(self, services, config):
        super().__init__(services, config)
        print('Created %s' % (self.__class__))

    def step(self, timestamp=0.0):
        print('Hello from hello_worker')
```

This *helloworld* package can be installed with

```
python -m pip install .
```

Or to install it in editable mode with

```
python -m pip install -e .
```

With the components installed as a package you can reference them by `MODULE` instead of providing the full path with `SCRIPT`. So to use the *hello_driver* you do `MODULE = helloworld.hello_driver`, and for *hello_worker* you can do `MODULE = helloworld.hello_worker`.

A simple config to run this is, `helloworld.config`

```
SIM_NAME = helloworld
SIM_ROOT = $PWD
LOG_FILE = log
LOG_LEVEL = INFO
SIMULATION_MODE = NORMAL

[PORTS]
    NAMES = DRIVER WORKER
    [[DRIVER]]
        IMPLEMENTATION = hello_world_driver

    [[WORKER]]
        IMPLEMENTATION = hello_world

[hello_world_driver]
    CLASS = driver
    SUB_CLASS =
    NAME = hello_driver
    NPROC = 1
    BIN_PATH =
    INPUT_FILES =
    OUTPUT_FILES =
    SCRIPT =
    MODULE = helloworld.hello_driver

[hello_world]
    CLASS = workers
    SUB_CLASS =
    NAME = hello_worker
```

(continues on next page)

(continued from previous page)

```
    NPROC = 1
    BIN_PATH =
    INPUT_FILES =
    OUTPUT_FILES =
    SCRIPT =
    MODULE = helloworld.hello_worker
```

And you need a platform file, `platform.conf`

```
MPIRUN = eval
NODE_DETECTION = manual
CORES_PER_NODE = 1
SOCKETS_PER_NODE = 1
NODE_ALLOCATION_MODE =  shared
HOST =
```

So after installing *ipsframework* and *helloworld* you can run it with

```
ips.py --config=helloworld.config --platform=platform.conf
```

and you should get the output

```
Created <class 'helloworld.hello_driver.hello_driver'>
Created <class 'helloworld.hello_worker.hello_worker'>
hello_driver: beginning step call
Hello from hello_worker
hello_driver: finished step call
```

3.5.1 Using PYTHONPATH instead of installing the package

If you don't want to install the package, this can still work if you set your `PYTHONPATH` correctly. In this case you don't need the `setup.py` either.

You can run the helloworld example from within the directory without installing by

```
PYTHONPATH=$PWD ips.py --config=helloworld.config --platform=platform.conf
```

3.6 Migrating from old IPS v0.1.0 to new IPS

This is a guide on converting from the old (up to July 2020) way of doing things to the new way.

The old version of IPS can be found at https://github.com/HPC-SimTools/IPS-framework/releases/tag/v0.1.0 and you can check it out by

```
git clone -b v0.1.0 https://github.com/HPC-SimTools/IPS-framework.git
```

IPS was originally run in a mode where either it was install into a directory with cmake or run from the source directory. The PYTHONPATH and PATH was set to point to the framework/src directory. Thing where imported directly from the modules.

Thing have changed, the package install is now managed with python setuptools and the IPS framework is install as a package called ipsframework, see *Building and Setting up Your Environment*. The ips.py executable is also installed

in you current PATH. This means that you no longer need to set PYTHONPATH or PATH when the IPS framework is installed. Likewise, there should also no longer be any need to reference the IPS framework using `IPS_ROOT` or `IPS_PATH`. This required a rearrangement of the source code.

Also with this change in the way the package is install there are required code changes need to use it. The main one is that since this is now a package everything must be imported from ipsframework, so when writing components you can no longer do `from component import Component` and must do `from ipsframework import Component`. Similarly if importing the framework directly you can not do `from ips import Framework` and now must do `from ipsframework import Framework`.

Additionally the following changes have been made

- These unused options have been remove from ips.py (`--component`, `--clone`, `--sim_name`, `--create-runspace`, `--run-setup`, `--run`, `--all`)
- A new option for components ports now allows you to specify a `MODULE` instead of a `SCRIPT`, this allows easy use of component that have been installed in the python environment.

These API have been deprecated for a long time and have been removed, you should update you code:

class	removed API	new API
ConfigurationManager	`getPort()`	*get_port()*
ServicesProxy	`getGlobalConfigParameter()`	*get_config_param()*
ServicesProxy	`getPort()`	*get_port()*
ServicesProxy	`getTimeLoop()`	*get_time_loop()*
ServicesProxy	`merge_current_plasma_state()`	*merge_current_state()*
ServicesProxy	`stage_plasma_state()`	*stage_state()*
ServicesProxy	`stageCurrentPlasmaState()`	*stage_state()*
ServicesProxy	`stageInputFiles()`	*stage_input_files()*
ServicesProxy	`stageOutputFiles()`	*stage_output_files()*
ServicesProxy	`update_plasma_state()`	*update_state()*
ServicesProxy	`updatePlasmaState()`	*update_state()*
ServicesProxy	`updateTimeStamp()`	*update_time_stamp()*

These simulation configuration fields have been deprecated for a long time and now have been remove, you should be update.

deprecated field	new field
`PLASMA_STATE_FILES`	`STATE_FILES`
`PLASMA_STATE_WORK_DIR`	`STATE_WORK_DIR`

The RUS (Resource Usage Simulator) has not been updated to python 3 or for the changes in IPS and will not function in it current state.

3.7 Installing IPS on NERSC

NERSC recommends the use of anaconda environments to manage python installs, see Brief introduction to Python at NERSC.

There is a conda environment already constructed and maintained for the *atom* project created using the *shareable environment* method. You can activate it and run IPS by:

```
module load python
source activate /global/common/software/atom/cori/ips-framework-new
ips.py --config=simulation.config --platform=platform.conf
```

3.7.1 Creating you own conda environment

This guide will go through creating a conda environment on NERSC and installing the IPS Framework using Option 2: Module + source activate

First, you need to load the python module, then create and activate a new conda environment. This will create the conda environment in your home directory (`$HOME/.conda/envs`):

```
module load python
conda create --name my_ips_env python=3.8 # or any version of python >=3.6
source activate my_ips_env
```

If you would like the same packages and versions in your conda environment as found in the python modules on Cori, you can clone that environment. In this case using `python/3.7-anaconda-2019.10`.

```
module load python/3.7-anaconda-2019.10
conda create -n my_ips_env --clone base
source activate my_ips_env
```

Next, install IPS-Framework into the conda environment

```
python -m pip install ipsframework
```

To leave your environment

```
conda deactivate
```

The example below shows how to select the newly create conda environment in a batch script, see Running Python in a batch job

```
#!/bin/bash
#SBATCH --constraint=haswell
#SBATCH --nodes=1
#SBATCH --time=5

module load python
source activate my_ips_env
ips.py --config=simulation.config --platform=platform.conf
```

3.7.2 Creating a shareable environment on /global/common/software

By default when you create a conda environment it will be created in `$HOME/.conda/envs`, to create one elsewhere that can be used by others you can use the `--prefix` option, see Creating conda environments.

In this example we are cloning the conda environment from the `python/3.7-anaconda-2019.10` module and install `ipsframework`.

```
module load python/3.7-anaconda-2019.10
conda create --prefix /global/common/software/myproject/env --clone base
source activate /global/common/software/myproject/env
python -m pip install ipsframework
```

The example below shows how to select the newly create conda environment in you batch script, see Running Python in a batch job

```
#!/bin/bash
#SBATCH --constraint=haswell
#SBATCH --nodes=1
#SBATCH --time=5

module load python
source activate /global/common/software/myproject/env
ips.py --config=simulation.config --platform=platform.conf
```

Installing dependencies

To see which packages are currently install in your environment run:

```
conda list
```

You can install any other dependencies you need by

```
conda install numpy matplotlib netcdf4 ...
```

User development

You should keep your development environment separate from the production environment. If you do development in your `my_ips_env` conda environment you can switch between that and the production environment on the atom project by

```
# switch to production environment
source activate /global/common/software/atom/cori/ips-framework-new

# switch bask to user development environment
source activate my_ips_env
```

Your bash prompt should be updated to reflect which environment you have active.

3.8 IPS Portal

The IPS portal hosted on the NERSC Spin service, shows the progress and status of IPS runs on a variety of machines. The simulation configuration file and platform configuration file contain entries that allow the IPS to publish events to the portal.

On the top-level page, you will see information about each run including who ran it, the current status, physics time stamp, wall time, and a descriptive comment. From there you can click on a Run ID to see the details of that run, including calls on components, data movement events, task launches and finishes, and checkpoints.

To use the portal include

```
USE_PORTAL = True
PORTAL_URL = http://lb.ipsportal.production.svc.spin.nersc.org
```

The source code for the portal can be found one GitHub and issues can be reported using GitHub issues.

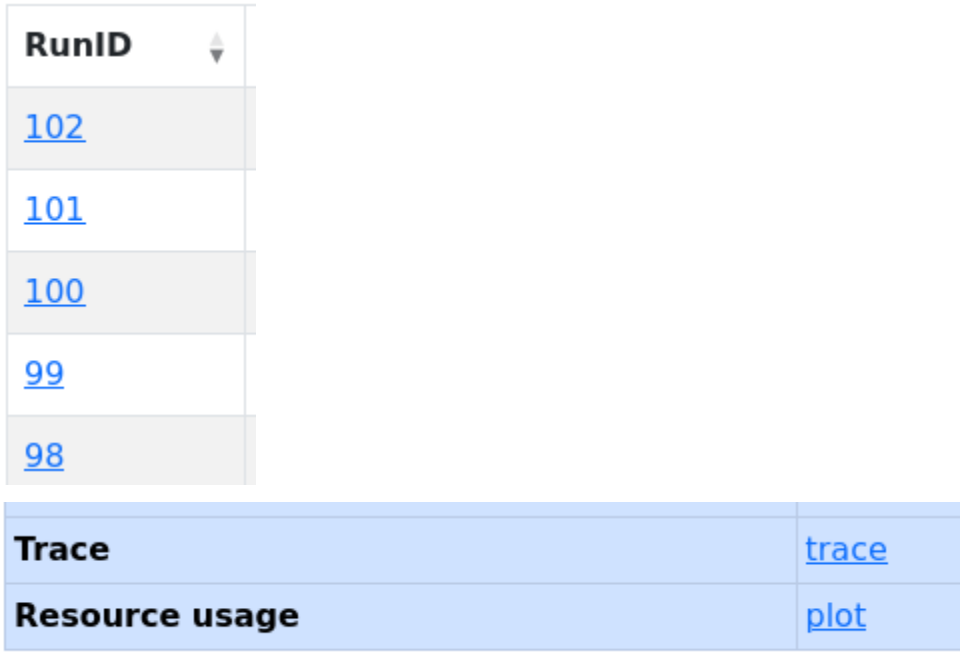
in either your *Platform Configuration File* or your *Simulation Configuration File*.

3.8.1 Tracing

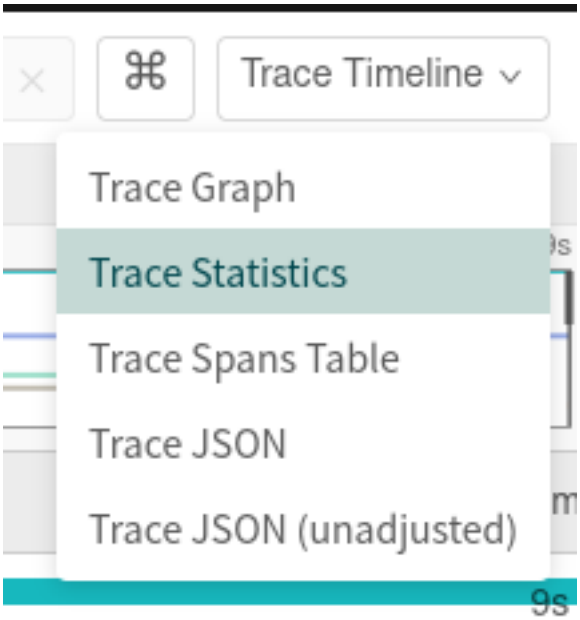
Note: New in IPS-Framework 0.6.0

IPS has the ability to capture a trace of the workflow to allow analysis and visualizations. The traces are captured in the Zipkin Span format and viewed within IPS portal using Jaeger.

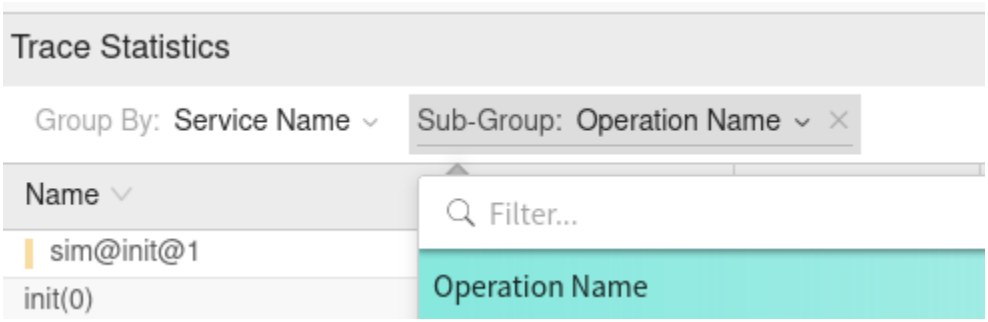
After selecting a run in the portal there will be a link to the trace:



The default view is the Trace Timeline but other useful views are Trace Graph and Trace Statistic which can be selected from the menu in the top-right:



The statistics can be further broken down by operation.



Note: Self time (ST) is the total time spent in a span when it was not waiting on children. For example, a 10ms span with two 4ms non-overlapping children would have self-time = 10ms - 2 * 4ms = 2ms.

3.8.2 Child Runs

Note: New in IPS-Framework 0.7.0

If you have a workflow where you are running `ips` as a task of another IPS simulation you can create a relation between them that will allow it to be viewed together in the IPS-portal and get a single trace for the entire collection.

To setup the hierarchical structure between different IPS runs, so if one run starts other runs as a separate simulation, you can set the `PARENT_PORTAL_RUNID` parameter in the child simulation configuration. This can be done dynamically from the parent simulation like:

```
child_conf['PARENT_PORTAL_RUNID'] = self.services.get_config_param("PORTAL_RUNID")
```

This is automatically configured when running `ips_dakota_dynamic.py`.

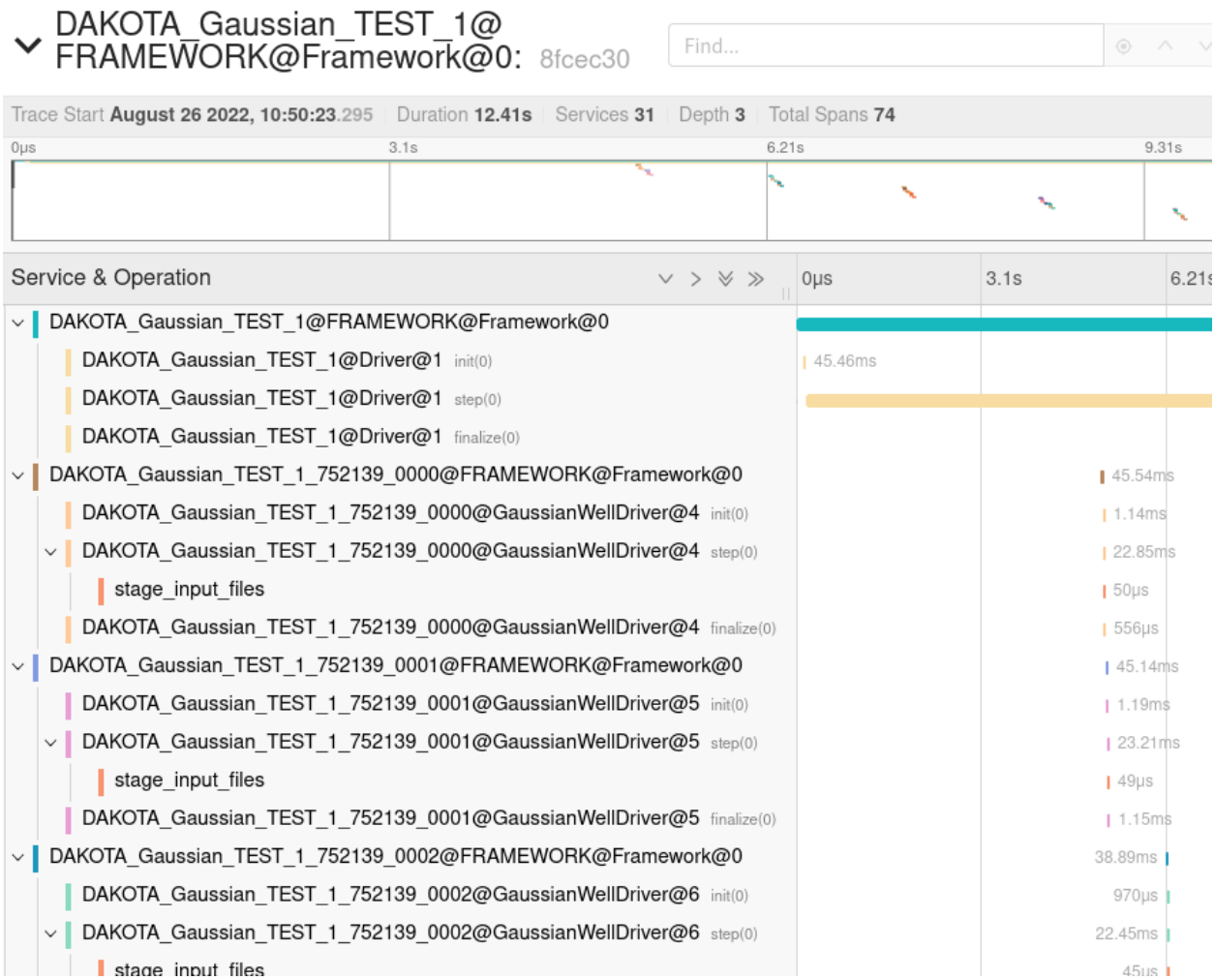
The child runs will not appear on the main runs list but will appear on a tab next to the events.

Events | Child runs

Reload | Search:

RunID	Status	Comment	Sim Name	Host	User	Start Time	Stop Time	Walltime
581	Completed	Testing dakota	DAKOTA_Gaussian_TEST_1_752139_0013	workstation	rwp	2022-08-26|10:50:35EDT	2022-08-26|10:50:35EDT	0.06
580	Completed	Testing dakota	DAKOTA_Gaussian_TEST_1_752139_0012	workstation	rwp	2022-08-26|10:50:35EDT	2022-08-26|10:50:35EDT	0.04
579	Completed	Testing dakota	DAKOTA_Gaussian_TEST_1_752139_0011	workstation	rwp	2022-08-26|10:50:33EDT	2022-08-26|10:50:33EDT	0.05

The trace of the primary simulation will contain the traces from all the simulations:



3.9 Dask

The ability to use Dask for task pool scheduling has been added and can be used by setting `use_dask=True` in *`submit_tasks()`*.

You can decide how many nodes to use by setting `dask_nodes`, one Dask worker will be created on every node, Dask will always use an entire node, using all cores on the node.

The workflow added to IPS using Dask allows for more than just running binary executables, you can run python functions and class methods.

An example showing this is the following, where we are adding an executable (in this case `sleep`), a function that sleeps (`myFun`) and a method that sleeps (`myMethod`) respectively to a task pool and submitting the task pool with `self.services.submit_tasks('pool', use_dask=True)`.

The `driver.py` in the most simplest form would be

```
from ipsframework import Component


class Driver(Component):
    def step(self, timestamp=0.0):
        worker_comp = self.services.get_port('WORKER')
        self.services.call(worker_comp, 'step', 0.0)
```

And the `dask_worker.py` is

```
import copy
from time import sleep
from ipsframework import Component


def myFun(*args):
    print(f"myFun({args[0]})")
    sleep(float(args[0]))
    return 0


class DaskWorker(Component):
    def step(self, timestamp=0.0):
        cwd = self.services.get_working_dir()
        self.services.create_task_pool('pool')

        duration = 0.5
        self.services.add_task('pool', 'binary', 1, cwd, self.EXECUTABLE, duration)
        self.services.add_task('pool', 'function', 1, cwd, myFun, duration)
        self.services.add_task('pool', 'method', 1, cwd, copy.copy(self).myMethod,␣
↪duration)

        ret_val = self.services.submit_tasks('pool',
                                             use_dask=True,
                                             dask_nodes=1)
        print('ret_val =', ret_val)
        exit_status = self.services.get_finished_tasks('pool')
        print('exit_status = ', exit_status)
```

(continues on next page)

(continued from previous page)

```
    def myMethod(self, *args):
        print(f"myMethod({args[0]})")
        sleep(float(args[0]))
        return 0
```

A simple config to run this is, `dask_sim.config`

```
SIM_NAME = dask_example
SIM_ROOT = $PWD
LOG_FILE = log
LOG_LEVEL = INFO
SIMULATION_MODE = NORMAL

[PORTS]
    NAMES = DRIVER WORKER
    [[DRIVER]]
        IMPLEMENTATION = driver

    [[WORKER]]
        IMPLEMENTATION = dask_worker

[driver]
    CLASS = DRIVER
    SUB_CLASS =
    NAME = Driver
    NPROC = 1
    BIN_PATH =
    INPUT_FILES =
    OUTPUT_FILES =
    SCRIPT = $PWD/driver.py

[dask_worker]
    CLASS = DASK_WORKER
    SUB_CLASS =
    NAME = DaskWorker
    NPROC = 1
    BIN_PATH =
    INPUT_FILES =
    OUTPUT_FILES =
    SCRIPT = $PWD/dask_worker.py
    EXECUTABLE = $PWD/sleep
```

This is executed with `ips.py --config dask_sim.config --platform platform.conf` and the output shows each different task type executing:

```
...
ret_val = 3
myFun(0.5)
myMethod(0.5)
/bin/sleep 0.5
exit_status =  {'binary': 0, 'method': 0, 'function': 0}
...
```

The output simulation log includes the start and end time of each task with in the pool with the elapsed time as expected, a trimmed JSON simulation log is shown:


```
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_LAUNCH_DASK_TASK",
  "walltime": "2.33",
  "comment": "task_name = method, Target = myMethod(0.5)",
}
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_LAUNCH_DASK_TASK",
  "walltime": "2.33",
  "comment": "task_name = function, Target = myFun(0.5)",
}
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_LAUNCH_DASK_TASK",
  "walltime": "2.33",
  "state": "Running",
  "comment": "task_name = binary, Target = sleep 0.5",
}
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_TASK_END",
  "walltime": "2.83",
  "comment": "task_name = method, elapsed time = 0.50s",
}
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_TASK_END",
  "walltime": "2.83",
  "comment": "task_name = function, elapsed time = 0.50s",
}
{
  "code": "DASK_WORKER__DaskWorker",
  "eventtype": "IPS_TASK_END",
  "walltime": "2.85",
  "state": "Running",
  "comment": "task_name = binary, elapsed time = 0.52s",
}
```


3.9.1 Running dask in shifter

Shifter is a resource for running docker containers on HPC. Documentation can be found here.

An option *use_shifter* has been added to *`submit_tasks()`* that will run the Dask scheduler and workers run inside the shifter container. Additional arguments to be passed to shifter when launching dask (such as *–image* and *–module*) can be set by the *shifter_args* parameters,

You will need to match the versions of Dask within the shifter container to the version running outside. This is because the Dask scheduler and workers run inside the container while IPS has the sk client outside.

As an example would be using the module `python/3.8-anaconda-2020.11` and the docker image `continuumio/`

`anaconda3:2020.11` which will have the same environment.

You will need to have IPS installed in the conda environment `python -m pip install ipsframework`. IPS is not required inside the shifter container, only the Dask scheduler and workers are running inside.

To pull down the docker into shifter run:

```
shifterimg pull continuumio/anaconda3:2020.11
```

You can entry the shifter container and check it's contents with:

```
shifter --image=continuumio/anaconda3:2020.11 /bin/bash
```

You batch script should then look like:

```
#!/bin/bash
...
#SBATCH --image=continuumio/anaconda3:2020.11

module load python/3.8-anaconda-2020.11

ips.py --config=ips.conf --platform=platform.conf
```

3.9.2 Running with worker plugin

There is the ability to set a `WorkerPlugin` on the dask worker using the *dask_worker_plugin* option in *submit_tasks()*.

Using a WorkerPlugin in combination with shifter allows you to do things like coping files out of the Temporary XFS file system. An example of that is

```
from distributed.diagnostics.plugin import WorkerPlugin

class CopyWorkerPlugin(WorkerPlugin):
    def __init__(self, tmp_dir, target_dir):
        self.tmp_dir = tmp_dir
        self.target_dir = target_dir

    def teardown(self, worker):
        os.system(f"cp {self.tmp_dir}/* {self.target_dir}")

class Worker(Component):
    def step(self, timestamp=0.0):
        cwd = self.services.get_working_dir()
        tmp_xfs_dir = '/tmp'

        self.services.create_task_pool('pool')
        self.services.add_task('pool', 'task_1', 1, tmp_xfs_dir, 'executable')

        worker_plugin = CopyWorkerPlugin(tmp_xfs_dir, cwd)

        ret_val = self.services.submit_tasks('pool',
                                             use_dask=True, use_shifter=True,
                                             dask_worker_plugin=worker_plugin)
```

(continues on next page)

(continued from previous page)

```
        exit_status = self.services.get_finished_tasks('pool')
```

where the batch script has the temporary XFS filesystem mounted as

```
#SBATCH --volume="/global/cscratch1/sd/$USER/tmpfiles:/tmp:perNodeCache=size=1G"
```

Continuous Archiving

Another example is a WorkerPlugin that will continuously create a tar archive of the output data at a regular interval while tasks are executing. This is useful should the workflow fail or is canceled before everything is finished. It creates a separate achieve for each node/worker since the temporary XFS filesystem is unique per node. This example creates an archive of all the data in the working directory every 60 seconds and again when everything is finished.

```
def file_daemon(worker_id, evt, source_dir, target_dir):
    cmd = f"tar -caf {target_dir}/{worker_id}_archive.tar.gz -C {source_dir} ."

    while not evt.wait(60):  # interval which to archive data
        os.system(cmd)

    os.system(cmd)

class ContinuousArchivingWorkerPlugin(WorkerPlugin):
    def __init__(self, tmp_dir, target_dir):
        self.tmp_dir = tmp_dir
        self.target_dir = target_dir

    def setup(self, worker):
        self.evt = Event()
        self.thread = Thread(target=file_daemon, args=(worker.id, self.evt, self.tmp_dir,
↪ self.target_dir))
        self.thread.start()

    def teardown(self, worker):
        self.evt.set()  # tells the thread to exit
        self.thread.join()

class Worker(Component):
    def step(self, timestamp=0.0):
        cwd = self.services.get_working_dir()
        tmp_xfs_dir = '/tmp'

        self.services.create_task_pool('pool')
        self.services.add_task('pool', 'task_1', 1, tmp_xfs_dir, 'executable')

        worker_plugin = ContinuousArchivingWorkerPlugin(tmp_xfs_dir, cwd)

        ret_val = self.services.submit_tasks('pool',
                                             use_dask=True, use_shifter=True,
                                             dask_worker_plugin=worker_plugin)
```

(continues on next page)

(continued from previous page)

```
        exit_status = self.services.get_finished_tasks('pool')
```

where the batch script has the temporary XFS filesystem mounted as

```
#SBATCH --volume="/global/cscratch1/sd/$USER/tmpfiles:/tmp:perNodeCache=size=1G"
```

CHAPTER

FOUR

DEVELOPER GUIDE

This document is for the development of IPS itself, if you want to develop drivers and components for IPS simulations see *The IPS for Driver and Component Developers*.

4.1 Contributing

You can report bugs (including security bugs) using GitHub issues.

Alternatively the developers can be contacted at discussions.

Change requests can be made using GitHub pull request.

4.1.1 Getting and installing IPS from source code

To get started you first need to obtain the source code, I suggest installing in editable mode, see *Installing IPS from source*.

4.1.2 Development environment

IPS-framework doesn't have any required dependencies. It has an optional dependency Dask that will enable Dask to be used for task pool scheduling, see `submit_tasks()`.

IPS-framework will work with python version 3.6. It is tested to work with Dask and distributed 2.5.2 but may work with earlier versions.

IPS-framework will work on Linux and macOS. It won't work on Windows directly but will work in the Windows Subsystem for Linux.

To run the tests requires `pytest`, `pytest-cov` and `psutil`. Optional dependencies are `dask/distributed` and `mpirun/mpi4py` which are needed to run all the tests.

It is recommend that you use conda but you also just install dependencies using system packages or with PyPI in an virtual environment.

Conda

To create a Conda environment with all testing dependencies run:

```
conda create -n ips python=3.8 pytest pytest-cov psutil dask mpi4py sphinx
conda activate ips
```

4.2 Code review expectations

Code will need to conform to the style as enforced by flake8 and should not introduce any new warnings or error from the static analysis, see *Static Analysis*.

All new features should have an accompanying test where it should try to include complete code coverage of the changes, see *Testing*.

All new functionality should have complete docstrings. If appropriate, further documentation or usage examples should be added, see *Documentation*.

4.3 Testing

4.3.1 Running Tests

The pytest framework is used for finding and executing tests in IPS-framework.

To run the tests

```
python -m pytest
```

To run test showing code coverage, install *pytest-cov* and run

```
python -m pytest --cov
```

and the output will look like

```
----------- coverage: platform linux, python 3.7.8-final-0 -----------
Name                                      Stmts   Miss  Cover
-------------------------------------------------------------
ipsframework/__init__.py                     11      0   100%
ipsframework/cca_es_spec.py                  62     10    84%
ipsframework/component.py                   105     19    82%
ipsframework/componentRegistry.py           105     25    76%
ipsframework/configurationManager.py        510    103    80%
ipsframework/convert_log_function.py         29      1    97%
ipsframework/dataManager.py                  72     15    79%
ipsframework/debug.py                         3      0   100%
ipsframework/eventService.py                137     53    61%
ipsframework/eventServiceProxy.py           118     49    58%
ipsframework/ips.py                         360     51    86%
ipsframework/ipsExceptions.py                61      2    97%
ipsframework/ipsLogging.py                   92      8    91%
ipsframework/ips_es_spec.py                  43      7    84%
```

(continues on next page)

(continued from previous page)

```
ipsframework/ipsutil.py                      73     26    64%
ipsframework/messages.py                     58      0   100%
ipsframework/node_structure.py              193     31    84%
ipsframework/platformspec.py                 18      4    78%
ipsframework/portalBridge.py                205     36    82%
ipsframework/resourceHelper.py              304     59    81%
ipsframework/resourceManager.py             340     69    80%
ipsframework/runspaceInitComponent.py        88     31    65%
ipsframework/sendPost.py                     41      2    95%
ipsframework/services.py                   1200    234    80%
ipsframework/taskManager.py                 322     74    77%
ipsframework/topicManager.py                 59      5    92%
------------------------------------------------------------
TOTAL                                      4609    914    80%
```

You can then also run `python -m coverage report -m` to show exactly which lines are missing test coverage.

4.3.2 Cori only tests

The are some tests that only run on Cori at NERSC and these are not run as part of the *CI* and must be run manually. To run those test you need to add the option `--runcori` to the `pytest`. There are tests for the *shifter functionally* that is Cori specific. There are also tests for the srun commands built with different `task_ppn` and `task_cpp` options in *launch_task()*.

An example batch script for running the unit tests is:

```
#!/bin/bash
#SBATCH -p debug
#SBATCH --nodes=1
#SBATCH -t 00:10:00
#SBATCH -C haswell
#SBATCH -J pytest
#SBATCH -e pytest.err
#SBATCH -o pytest.out
#SBATCH --image=continuumio/anaconda3:2020.11
module load python/3.8-anaconda-2020.11
python -m pytest --runcori
```

Then check the output in `pytest.out` to see that all the tests passed.

4.3.3 Perlmutter only tests

The are some tests that only run on Perlmutter at NERSC and these are not run as part of the *CI* and must be run manually. To run those test you need to add the option `--runperlmutter` to the `pytest`.There are also tests for the srun commands built with different `task_ppn`, `task_cpp` and `task_gpp` options in *launch_task()*.

An example batch script for running the unit tests is:

```
#!/bin/bash
#SBATCH -p debug
#SBATCH --nodes=1
#SBATCH -t 00:20:00
```

(continues on next page)

(continued from previous page)

```
#SBATCH -C gpu
#SBATCH -J pytest
#SBATCH -e pytest.err
#SBATCH -o pytest.out
module load python
python -m pytest --runperlmutter
```

Then check the output in `pytest.out` to see that all the tests passed.

4.3.4 Writing Tests

The pytest framework is used for finding and executing tests in IPS-framework.

Tests should be added to `tests` directory. If writing component to use for testing that should go into `tests/components` and any executable should go into `tests/bin`.

4.4 Continuous Integration (CI)

GitHub Actions is used for CI and will run on all pull requests and any branch including once a pull request is merged into `main`. Static analysis checks and the test suite will run and report the code coverage to Codecov.

4.4.1 Static Analysis

The following static analysis is run as part of CI

- flake8 - Style guide enforcement
- pylint - Code analysis
- bandit - Find common security issues
- codespell - Check code for common misspellings

The configuration of these tools can be found in setup.cfg.

4.4.2 Tests

The test suite runs on Linux and macOS with python versions from 3.6 up to 3.9. It is also tested with 3 different version of Dask, `2.5.2`, `2.30.0` and the most recent version. The `2.5.2`, `2.30.0` versions of Dask where chosen to match what is available on Cori at NERSC in the modules `python/3.7-anaconda-2019.10` and `python/3.8-anaconda-2020.11`.

The test suite also runs as part of the CI on Windows using WSL (Ubuntu 20.04) just using the default system python version.

4.5 Documentation

sphinx is used to generate the documentation for IPS. The docs are found in the `doc` directory and the docstrings from the source code can included in the documentation. The documentation can be built by running `make html` within the `doc` directory, the output will go to `doc/_build/html`.

The docs are automatically build by Read the Docs when merged into `main` and deployed to http://ips-framework.readthedocs.io. You can see the status of the docs build by going to here

4.6 Release process

We have no set release schedule and will create minor (add functionality in a backwards compatible manner) and patch (bug fixes) releases as needed following Semantic Versioning.

The deployment to PyPI will happen automatically by a GitHub Actions workflow whenever a tag is created.

Release notes should be added to https://github.com/HPC-SimTools/IPS-framework/releases

We will publish a release candidate versions for any major or minor release before the full release to allow feedback from users. Patch versions will not normally have an release candidate.

Before a release is finalized the *Cori only tests* and *Perlmutter only tests* should be run.

CHAPTER

FIVE

CODE LISTINGS

5.1 IPS

The Integrated Plasma Simulator (IPS) Framework. This framework enables loose, file-based coupling of certain class of nuclear fusion simulation codes.

For further design information see

- Wael Elwasif, David E. Bernholdt, Aniruddha G. Shet, Samantha S. Foley, Randall Bramley, Donald B. Batchelor, and Lee A. Berry, *The Design and Implementation of the SWIM Integrated Plasma Simulator*, in The 18th Euromirco International Conference on Parallel, Distributed and Network - Based Computing (PDP 2010), 2010.
- Samantha S. Foley, Wael R. Elwasif, David E. Bernholdt, Aniruddha G. Shet, and Randall Bramley, *Extending the Concept of Component Interfaces: Experience with the Integrated Plasma Simulator*, in Component - Based High - Performance Computing (CBHPC) 2009, 2009, (extended abstract).
- D Batchelor, G Alba, E D'Azevedo, G Bateman, DE Bernholdt, L Berry, P Bonoli, R Bramley, J Breslau, M Chance, J Chen, M Choi, W Elwasif, S Foley, G Fu, R Harvey, E Jaeger, S Jardin, T Jenkins, D Keyes, S Klasky, S Kruger, L Ku, V Lynch, D McCune, J Ramos, D Schissel, D Schnack, and J Wright, *Advances in Simulation of Wave Interactions with Extended MHD Phenomena*, in Horst Simon, editor, SciDAC 2009, 14-18 June 2009, San Diego, California, USA, volume 180 of Journal of Physics: Conference Series, page 012054, Institute of Physics, 2009, 6pp.
- Samantha S. Foley, Wael R. Elwasif, Aniruddha G. Shet, David E. Bernholdt, and Randall Bramley, *Incorporating Concurrent Component Execution in Loosely Coupled Integrated Fusion Plasma Simulation*, in Component-Based High-Performance Computing (CBHPC) 2008, 2008, (extended abstract).
- D. Batchelor, C. Alba, G. Bateman, D. Bernholdt, L. Berry, P. Bonoli, R. Bramley, J. Breslau, M. Chance, J. Chen, M. Choi, W. Elwasif, G. Fu, R. Harvey, E. Jaeger, S. Jardin, T. Jenkins, D. Keyes, S. Klasky, S. Kruger, L. Ku, V. Lynch, D. McCune, J. Ramos, D. Schissel, D. Schnack, and J. Wright, *Simulation of Wave Interactions with MHD*, in Rick Stevens, editor, SciDAC 2008, 14-17 July 2008, Washington, USA, volume 125 of Journal of Physics: Conference Series, page 012039, Institute of Physics, 2008.
- Wael R. Elwasif, David E. Bernholdt, Lee A. Berry, and Don B. Batchelor, *Component Framework for Coupled Integrated Fusion Plasma Simulation*, in HPC-GECO/CompFrame 2007, 21-22 October, Montreal, Quebec, Canada, 2007.

Authors Wael R. Elwasif, Samantha Foley, Aniruddha G. Shet

Organization Center for Simulation of RF Wave Interactions with Magnetohydrodynamics

5.2 Framework

class ipsframework.ips.**Framework**(*config_file_list*, *log_file_name*, *platform_file_name=None*, *debug=False*, *verbose_debug=False*, *cmd_nodes=0*, *cmd_ppn=0*)

Create an IPS Framework Instance to coordinate the execution of IPS simulations

The Framework performs the following main tasks:

- Initialize the different IPS managers that perform the bulk of the framework functionality
- Manage communication queues, and route service requests from simulation components to appropriate managers.
- Provide logging services to IPS managers.
- Perform shutdown procedure on exit

Parameters

- **config_file_list** (*list*) – A list of simulation configuration files to be used in the simulaion. Each simulation configuration file must have the following parameters
  - *SIM_ROOT* The root directory for the simulation
  - *SIM_NAME* A name that identifies the simulation
  - *LOG_FILE* The name of a log file that is used to capture logging and error information for this simulation.

  SIM_ROOT, *SIM_NAME*, and *LOG_FILE* must be unique across simulations.
- **log_file_name** (*str*) – A file name where Framework logging messages are placed.
- **platform_file_name** (*str*) – The name of the platform configuration file used in the simulation. If not specified it will try to find the one installed in the share directory.
- **debug** (*bool*) – A flag indicating whether framework debugging messages are enabled (default = False)
- **verbose_debug** (*bool*) – A flag adding more verbose framework debugging (default = False)
- **cmd_nodes** (*int*) – Computer nodes (default = 0)
- **cmd_ppn** (*int*) – Computer processor per nodes (default = 0)

critical(*msg*, **args*)

Produce **critical** message in simulation log file. See `logging.critical()` for usage.

debug(*msg*, **args*)

Produce **debugging** message in simulation log file. See `logging.debug()` for usage.

error(*msg*, **args*)

Produce **error** message in simulation log file. See `logging.error()` for usage.

exception(*msg*, **args*)

Produce **exception** message in simulation log file. See `logging.exception()` for usage.

get_inq()

Returns handle to the Framework's input queue object

Return type `multiprocessing.Queue`

info(*msg*, **args*)
Produce **informational** message in simulation log file. See `logging.info()` for usage.

initiate_new_simulation(*sim_name*)
This is to be called by the configuration manager as part of dynamically creating a new simulation. The purpose here is to initiate the method invocations for the framework-visible components in the new simulation

log(*msg*, **args*)
Wrapper for *Framework.info()*.

register_service_handler(*service_list*, *handler*)
Register a call back method to handle a list of framework service invocations.

Parameters

- **service_list** – a list of service names to call *handler* when invoked by components. The service name must match the *target_method* parameter in `messages.ServiceRequestMessage`.
- **handler** – a Python callable object that takes a `messages.ServiceRequestMessage`.

run()
Run the communication outer loop of the framework.

This method implements the core communication and message dispatch functionality of the framework. The main phases of execution for the framework are:

1. Invoke the `init` method on all framework-attached components, blocking pending method call termination.
2. Generate method invocation messages for the remaining public method in the framework-centric components (i.e. `step` and `finalize`).
3. Generate a queue of method invocation messages for all public framework accessible components in the simulations being run. framework-accessible components are made up of the **Init** component (if is exists), and the **Driver** component. The generated messages invoke the public methods `init`, `step`, and `finalize`.
4. Dispatch method invocations for each framework-centric component and physics simulation in order.

Exceptions that propagate to this method from the managed simulations causes the framework to abort any pending method invocation for the source simulation. Exceptions from framework-centeric component aborts further invocations to that component.

When all method invocations have been dispatched (or aborted), `configurationManager.ConfigurationManager.terminate_sim()` is called to trigger normal termination of all component processes.

Returns Success status

Return type bool

send_terminate_msg(*sim_name*, *status=0*)
This method remotely invokes the method `component.Component.terminate()` on all componnets in the IPS simulation `sim_name`.

Parameters

- **sim_name** (*str*) – The simulation name from which all the components are terminated
- **status** (`messages.Message.SUCCESS`, `messages.Message.FAILURE`) – message status, defaults to `messages.Message.SUCCESS`

terminate_all_sims(*status=0*)
Terminate all active component instances.

This method remotely invokes the method `component.Component.terminate()` on all componnets in the IPS simulation.

Parameters **status** (`messages.Message.SUCCESS`, `messages.Message.FAILURE`) – message status, defaults to `messages.Message.SUCCESS`

warning(*msg*, **args*)
Produce **warning** message in simulation log file. See `logging.warning()` for usage.

5.3 Data Manager

class ipsframework.dataManager.**DataManager**(*fwk*)
The data manager facilitates the movement and exchange of data files for the simulation.

merge_current_plasma_state(*msg*)
Merge partial plasma state file with global master. Newly updated plasma state copied to caller's workdir. Exception raised on copy error.

msg.args:

0. partial_state_file
1. target_state_file
2. log_file: stdout for merge process if not `None`

process_service_request(*msg*)
Invokes the appropriate public data manager method for the component specified in *msg*. Return method's return value.

stage_state(*msg*)
Copy plasma state files from source dir to target dir. Return 0. Exception raised on copy error.

msg.args:

0. state_files
1. source_dir
2. target_dir

update_state(*msg*)
Copy plasma state files from source dir to target dir. Return 0. Exception raised on copy error.

msg.args:

0. state_files
1. source_dir
2. target_dir

5.4 Task Manager

class ipsframework.taskManager.**TaskInit**(*nproc*, *binary*, *working_dir*, *tppn*, *tcpp*, *tgpp*, *block*, *omp*, *wnodes*, *wsocks*, *cmd_args*, *launch_cmd_extra_args*)

binary
Alias for field number 1

block
Alias for field number 6

cmd_args
Alias for field number 10

launch_cmd_extra_args
Alias for field number 11

nproc
Alias for field number 0

omp
Alias for field number 7

tcpp
Alias for field number 4

tgpp
Alias for field number 5

tppn
Alias for field number 3

wnodes
Alias for field number 8

working_dir
Alias for field number 2

wsocks
Alias for field number 9

class ipsframework.taskManager.**TaskManager**(*fwk*)
The task manager is responsible for facilitating component method invocations, and the launching of tasks.

build_launch_cmd(*nproc*, *binary*, *cmd_args*, *working_dir*, *ppn*, *max_ppn*, *nodes*, *accurateNodes*, *partial_nodes*, *task_id*, *cpp=0*, *omp=False*, *gpp=0*, *core_list=''*, *launch_cmd_extra_args=None*)
Construct task launch command to be executed by the component.

- nproc - number of processes to use
- binary - binary to launch
- cmd_args - additional command line arguments for the binary
- working_dir - full path to where the executable will be launched
- ppn - processes per node value to use
- max_ppn - maximum possible ppn for this allocation
- nodes - comma separated list of node ids

- accurateNodes - if `True`, launch on nodes in *nodes*, otherwise the parallel launcher determines the process placement
- **partial_nodes - if `True` and *accurateNodes* and *task_launch_cmd* == 'mpirun',** a host file is created specifying the exact placement of processes on cores.
- core_list - used for creating host file with process to core mappings

`finish_task`(*finish_task_msg*)
Cleanup after a task launched by a component terminates

finish_task_msg is expected to be of type `messages.ServiceRequestMessage`

Message args:

0. *task_id*: task id of finished task
1. *task_data*: return code of task

`get_call_id`()
Return a new call id

`get_task_id`()
Return a new task id

`init_call`(*init_call_msg*, *manage_return=True*)
Creates and sends a `messages.MethodInvokeMessage` from the calling component to the target component. If *manage_return* is `True`, a record is added to *outstanding_calls*. Return call id.

Message args:

0. method_name
1. + arguments to be passed on as method arguments.

`init_task`(*init_task_msg*)
Allocate resources needed for a new task and build the task launch command using the binary and arguments provided by the requesting component. Return launch command to component via `messages.ServiceResponseMessage`. Raise exception if task can not be launched at this time (`ipsExceptions.BadResourceRequestException`, `ipsExceptions.InsufficientResourcesException`).

init_task_msg is expected to be of type `messages.ServiceRequestMessage`

Message args:

0. *nproc*: number of processes the task needs
1. *binary*: full path to the executable to launch

SIMYAN: added this to deal with the component directory change 2. *working_dir*: full path to directory where the task will be launched

3. *tppn*: processes per node for this task. (0 indicates that the default ppn is used.)
4. *block*: whether or not to wait until the task can be launched.
5. *wnodes*: `True` for whole node allocation, `False` otherwise.
6. *wsocks*: `True` for whole socket allocation, `False` otherwise.
7. + *cmd_args*: any arguments for the executable

`init_task_pool`(*init_task_msg*)
Allocate resources needed for a new task and build the task launch command using the binary and arguments provided by the requesting component.

init_task_msg is expected to be of type `messages.ServiceRequestMessage`

Message args:

0. *task_dict*: dictionary of task names and objects

initialize(*data_mgr*, *resource_mgr*, *config_mgr*)
Initialize references to other managers and key values from configuration manager.

printCurrTaskTable()
Prints the task table pretty-like.

process_service_request(*msg*)
Invokes the appropriate public data manager method for the component specified in *msg*. Return method's return value.

return_call(*response_msg*)
Handle the response message generated by a component in response to a method invocation on that component.

reponse_msg is expected to be of type `messages.MethodResultMessage`

wait_call(*wait_msg*)
Determine if the call has finished. If finished, return any data or errors. If not finished raise the appropriate blocking or nonblocking exception and try again later.

wait_msg is expected to be of type `messages.ServiceRequestMessage`

Message args:

0. *call_id*: call id for which to wait
1. *blocking*: determines the wait is blocking or not

5.5 Resource Manager

class ipsframework.resourceManager.**Allocation**(*partial_node*, *nodelist*, *corelist*, *ppn*, *max_ppn*, *cpp*, *accurateNodes*, *cores_allocated*)

accurateNodes
Alias for field number 6

corelist
Alias for field number 2

cores_allocated
Alias for field number 7

cpp
Alias for field number 5

max_ppn
Alias for field number 4

nodelist
Alias for field number 1

partial_node
Alias for field number 0

ppn
Alias for field number 3

class ipsframework.resourceManager.**ResourceManager**(*fwk*)

The resource manager is responsible for detecting the resources allocated to the framework, allocating resources to task requests, and maintaining the associated bookkeeping.

add_nodes(*listOfNodes*)

Add node entries to `self.nodes`. Typically used by *initialize()* to initialize `self.nodes`. May be used to add nodes to a dynamic allocation in the future.

listOfNodes is a list of tuples (*node name*, *cores*). `self.nodes` is a dictionary where the keys are the *node names* and the values are `node_structure.Node` structures.

Return total number of cores.

begin_RM_report()

Print header information for resource usage reporting file.

check_core_cap(*nproc*, *ppn*)

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* without further restrictions.. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

check_gpus(*ppn*, *task_gpp*)

check_whole_node_cap(*nproc*, *ppn*)

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* and whole nodes. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

check_whole_sock_cap(*nproc*, *ppn*)

Determine if it is currently possible to allocate *nproc* processes with a ppn of *ppn* and whole sockets. Return `True` and list of nodes to use if successful. Return `False` and empty list if there are not enough available resources at this time, but it is possible to eventually satisfy the request. Exception raised if the request can never be fulfilled.

get_allocation(*comp_id*, *nproc*, *task_id*, *whole_nodes*, *whole_socks*, *task_ppn=0*, *task_cpp=0*, *task_gpp=0*)

Traverse available nodes to return:

If *whole_nodes* is `True`:

- *shared_nodes*: `False`
- *nodes*: list of node names
- *ppn*: processes per node for launching the task
- *max_ppn*: processes that can be launched
- *accurateNodes*: `True` if *nodes* uses the actual names of the nodes, `False` otherwise.

If *whole_nodes* is `False`:

- *shared_nodes*: `True`
- *nodes*: list of node names
- ***node_file_entries*: list of (node, corelist) tuples, where *corelist* is a list of core names.** Core names are integers from 0 to n-1 where n is the number of cores on a node.
- *ppn*: processes per node for launching the task
- *max_ppn*: processes that can be launched

- *accurateNodes*: `True` if *nodes* uses the actual names of the nodes, `False` otherwise.

Arguments:

- *nproc*: the number of requested processes (int)
- *comp_id*: component identifier, must be unique with respect to the framework (string)
- *task_id*: task identifier from TM (int)
- *method*: name of method (string)
- *task_ppn*: ppn for this task (optional) (int)

initialize(*dataMngr*, *taskMngr*, *configMngr*, *cmd_nodes=0*, *cmd_ppn=0*)
Initialize resource management structures, references to other managers (*dataMngr*, *taskMngr*, *configMngr*).

Resource information comes from the following in order of priority:

- command line specification (*cmd_nodes*, *cmd_ppn*)
- detection using parameters from platform config file
- manual settings from platform config file

The second two sources are obtained through `resourceHelper.getResourceList()`.

printRMState()
Print the node tree to `stdout`.

process_service_request(*msg*)

release_allocation(*task_id*, *status*)
Set resources allocated to task *task_id* to available. *status* is not used, but may be used to correlate resource failures to task failures and implement task relaunch strategies.

report_RM_status(*notes=''*)
Print current RM status to the reporting_file ("resource_usage") Entries consist of:

- time in seconds since beginning of time (__init__ of RM)
- # cores that are available
- # cores that are allocated
- % allocated cores
- # processes launched by task
- % cores used by processes
- notes (a description of the event that changed the resource usage)

sendEvent(*eventName*, *info*)
wrapper for constructing and publishing EM events

---

class ipsframework.node_structure.**Node**(*name*, *socks*, *cores*, *p*)
Models a node in the allocation.

- *name*: name of node, typically actual name from resource detection phase.
- *task_ids*, *owners*: identifiers for the tasks and components that are currently using the node.
- *allocated*, *available*: list of sockets that have cores allocated and available. A socket may appear in both lists if it is only partially allocated.

- *sockets*: list of sockets belonging to this node
- *avail_cores*: number of cores that are currently available.
- *total_cores*: total number of cores that can be allocated on this node.
- *status*: indicates if the node is 'UP' or 'DOWN'. Currently not used, all nodes are considered functional..

allocate(*whole_nodes*, *whole_sockets*, *tid*, *o*, *procs*)
Mark *procs* number of cores as allocated subject to the values of *whole_nodes* and *whole_sockets*. Return the number of cores allocated and their corresponding slots, a list of strings of the form:

<socket name>:<core name>

print_sockets(*fname=''*)
Pretty print of state of sockets.

release(*tid*, *o*)
Mark cores used by task *tid* and component *o* as available. Return the number of cores released.

class ipsframework.node_structure.**Socket**(*name*, *cps*, *coreids=()*)
Models a socket in a node.

- *name*: identifier for the socket
- *task_ids*, *owners*: identifiers for the tasks and components that are currently using the socket.
- *allocated*, *available*: lists of cores that are allocated and available.
- *cores*: list of `Core` objects belonging to this socket
- *avail_cores*: number of cores that are currently available.
- *total_cores*: total number of cores that can be allocated on this socket.

allocate(*whole*, *tid*, *o*, *num_procs*)
Mark *num_procs* cores as allocated subject to the value of *whole*. Return a list of strings of the form:

<socket name>:<core name>

print_cores(*fname=''*)
Pretty print of state of cores.

release(*tid*)
Mark cores that are allocated to task *tid* as available. Return number of cores set to available.

class ipsframework.node_structure.**Core**(*name*)
Models a core of a socket.

- *name*: name of core
- *is_available*: boolean value indicating the availability of the core.
- *task_id*, *owner*: identifiers of the task and component using the core.

allocate(*tid*, *o*)
Mark core as allocated.

release()
Mark core as available.

The Resource Helper file contains all of the code needed to figure out what host we are on and what resources we have. Taking this out of the resource manager will allow us to test it independent of the IPS.

ipsframework.resourceHelper.**getResourceList**(*services*, *host*, *partial_nodes=False*)
Using the host information, the resources are detected. Return list of (<node name>, <processes per node>), cores per node, sockets per node, processes per node, and `True` if the node names are accurate, `False` otherwise.

ipsframework.resourceHelper.**get_checkjob_info**()

ipsframework.resourceHelper.**get_pbs_info**()
Access info about allocation from PBS environment variables:

`PBS_NNODES` `PBS_NODEFILE`

ipsframework.resourceHelper.**get_qstat_jobinfo**()
Use `qstat -f $PBS_JOBID` to get the number of nodes and ppn of the allocation. Typically works on PBS systems.

ipsframework.resourceHelper.**get_qstat_jobinfo2**()
A second way to use `qstat -f $PBS_JOBID` to get the number of nodes and ppn of the allocation. Typically works on PBS systems.

ipsframework.resourceHelper.**get_slurm_info**()
Access environment variables set by Slurm to get the node names, tasks per node and number of processes.

`SLURM_NODELIST` `SLURM_TASKS_PER_NODE` or `SLURM_JOB_TASKS_PER_NODE` `SLURM_NPROC`

ipsframework.resourceHelper.**manual_detection**(*services*)
Use values listed in platform configuration file.

5.6 Component

IPS Framework Component

class ipsframework.component.**Component**(*services*, *config*)
Base class for all IPS components. Common set up, connection and invocation actions are implemented here.

Parameters

- **services** (*ServicesProxy*) – service proxy to communicate with framework
- **config** (*dict*) – configuration dictionary for this component

property args

property call_id

checkpoint(*timestamp=0.0*, ***keywords*)
Produce some default debugging information before the rest of the code is executed.

property component_id

property config

finalize(*timestamp=0.0*, ***keywords*)
Produce some default debugging information before the rest of the code is executed.

init(*timestamp=0.0*, ***keywords*)
Produce some default debugging information before the rest of the code is executed.

property method_name

restart(*timestamp=0.0*, ***keywords*)
Produce some default debugging information before the rest of the code is executed.

property services

property start_time

step(*timestamp=0.0*, ***keywords*)
Produce some default debugging information before the rest of the code is executed.

terminate(*status*)
Clean up services and call sys_exit.

5.7 Component Registry

class ipsframework.componentRegistry.ComponentID(*class_name*, *sim_name*)
Object to facilitate the creation, serialization and deserialization of component ids.

all_ids = {}

delimiter = '@'

static deserialize(*comp_id_string*)
Return the deserialized version of the component id.

get_class_name()
Return class name of component.

get_instance_name()
Return instance name of component id.

get_seq_num()
Return sequence number of component.

get_serialization()
Return serialization.

get_sim_name()
Return simulation name for the component.

seq_num = 0

class ipsframework.componentRegistry.ComponentRegistry(**args*, ***kwargs*)

class RegistryEntry(*svc_response_q*, *invocation_q*, *component_ref*, *services*, *config*)
Container for queues and references associated with a component.

addEntry(*component_id*, *svc_response_q*, *invocation_q*, *component_ref*, *services*, *config*)
Create a component registry entry for *component_id* and its associated queues, component ref, services and configuration information.

getComponentArtifact(*component_id*, *artifact*)
Return value of *artifact* in *component_id*'s registry entry.

getEntry(*component_id*)
Return a registry entry.

get_component_ids(*sim_name*)
Return all of the component ids associated with sim *sim_name*

removeEntry(*component_id*)

setComponentArtifact(*component_id*, *artifact*, *value*)
Set the value of *artifact* in *component_id*'s registry entry to *value*.

class ipsframework.componentRegistry.SingletonMeta

5.8 Configuration Manager

class ipsframework.configurationManager.**ConfigurationManager**(*fwk*, *config_file_list*, *platform_file_name*)

The configuration manager is responsible for paring the simulation and platform configuration files, creating the framework and simulation components, as well as providing an interface to accessing items from the configuration files (e.g., the time loop).

class SimulationData(*sim_name*, *start_time=1674857514.412142*)
Structure to hold simulation data stored into the sim_map entry in the configurationManager class

create_simulation(*sim_name*, *config_file*, *override*, *sub_workflow=False*)

get_all_simulation_components_map()

get_all_simulation_sim_root()

get_component_map()
Return a dictionary of simulation names and lists of component references. (May only be the driver, and init (if present)???)

get_config_parameter(*sim_name*, *param*)
Return value of *param* from simulation configuration file for *sim_name*.

get_framework_components()
Return list of framework components.

get_platform_parameter(*param*, *silent=False*)
Return value of platform parameter *param*. If *silent* is `False` (default) `None` is returned when *param* not found, otherwise an exception is raised.

get_port(*sim_name*, *port_name*)
Return a reference to the component from simulation *sim_name* implementing port *port_name*.

get_sim_names()
Return list of names of simulations.

get_sim_parameter(*sim_name*, *param*)
Return value of *param* from simulation configuration file for *sim_name*.

get_simulation_components(*sim_name*)

initialize(*data_mgr*, *resource_mgr*, *task_mgr*)
Parse the platform and simulation configuration files using the `ConfigObj` module. Create and initialize simulation(s) and their components, framework components and loggers.

process_service_request(*msg*)
Invokes public configuration manager method for a component. Return method's return value.

set_config_parameter(*sim_name*, *param*, *value*, *target_sim_name*)
Set the configuration parameter *param* to value *value* in *target_sim_name*. If *target_sim_name* is the framework, all simulations will get the change. Return *value*.

terminate(*status*)
Terminates all processes attached to the framework. *status* not used.

terminate_sim(*sim_name*)

5.9 Services

IPS Services

class ipsframework.services.**RunningTask**(*process*, *start_time*, *timeout*, *nproc*, *cores_allocated*, *command*, *binary*, *args*)

args
Alias for field number 7

binary
Alias for field number 6

command
Alias for field number 5

cores_allocated
Alias for field number 4

nproc
Alias for field number 3

process
Alias for field number 0

start_time
Alias for field number 1

timeout
Alias for field number 2

class ipsframework.services.**ServicesProxy**(*fwk*, *fwk_in_q*, *svc_response_q*, *sim_conf*, *log_pipe_name*)
The *ServicesProxy* object is responsible for marshalling invocations of framework services to the framework process using a shared queue. The queue is shared among all components in a simulation. The results from framework services invocations are received via another, component-specific "framework response" queue.

Create a new ServicesProxy object

Parameters

- **fwk** (*ipsframework.ips.Framework*) – Enclosing IPS simulation framework
- **fwk_in_q** (multiprocessing.Queue) – Framework input message queue - shared among all service objects
- **svc_response_q** (multiprocessing.Queue) – Service response message queue - one per service object.
- **sim_conf** (*dict*) – Simulation configuration dictionary, contains data from the simulation configuration file merged with the platform configuration file.
- **log_pipe_name** (*str*) – Name of logging pipe for use by the IPS logging daemon.

add_task(*task_pool_name*, *task_name*, *nproc*, *working_dir*, *binary*, **args*, ***keywords*)
Add task *task_name* to task pool *task_pool_name*. Remaining arguments are the same as in *ServicesProxy.launch_task()*.

call(*component_id*, *method_name*, **args*, ***keywords*)
Invoke method *method_name* on component *component_id* with optional arguments **args*. Will wait until call is finished. Return result from invoking the method.

Parameters

- **component_id** (*ComponentID*) – Component ID of requested component
- **method_name** (*str*) – component method to call, e.g. `init` or `step`

Returns service response message arguments

`call_nonblocking`(*component_id*, *method_name*, **args*, ***keywords*)

Invoke method *method_name* on component *component_id* with optional arguments **args*. Will not wait until finished.

Parameters

- **component_id** (*ComponentID*) – Component ID of requested component
- **method_name** (*str*) – component method to call, e.g. `init` or `step`

Returns call_id

Return type int

`checkpoint_components`(*comp_id_list*, *time_stamp*, *Force=False*, *Protect=False*)

Selectively checkpoint components in *comp_id_list* based on the configuration section *CHECKPOINT*. If *Force* is `True`, the checkpoint will be taken even if the conditions for taking the checkpoint are not met. If *Protect* is `True`, then the data from the checkpoint is protected from clean up. *Force* and *Protect* are optional and default to `False`.

The *CHECKPOINT_MODE* option controls determines if the components checkpoint methods are invoked.

Possible *MODE* options are:

ALL: Checkpint every time the call is made (equivalent to always setting Force =True)

WALLTIME_REGULAR: checkpoints are saved upon invocation of the service call `checkpoint_components()`, when a time interval greater than, or equal to, the value of the configuration parameter WALLTIME_INTERVAL had passed since the last checkpoint. A checkpoint is assumed to have happened (but not actually stored) when the simulation starts. Calls to `checkpoint_components()` before WALLTIME_INTERVAL seconds have passed since the last successful checkpoint result in a NOOP.

WALLTIME_EXPLICIT: checkpoints are saved when the simulation wall clock time exceeds one of the (ordered) list of time values (in seconds) specified in the variable WALLTIME_VALUES. Let [t_0, t_1, ..., t_n] be the list of wall clock time values specified in the configuration parameter WALLTIME_VALUES. Then checkpoint(T) = True if T >= t_j, for some j in [0,n] and there is no other time T_1, with T > T_1 >= T_j such that checkpoint(T_1) = True. If the test fails, the call results in a NOOP.

PHYSTIME_REGULAR: checkpoints are saved at regularly spaced "physics time" intervals, specified in the configuration parameter PHYSTIME_INTERVAL. Let PHYSTIME_INTERVAL = PTI, and the physics time stamp argument in the call to checkpoint_components() be pts_i, with i = 0, 1, 2, ... Then checkpoint(pts_i) = True if pts_i >= n PTI , for some n in 1, 2, 3, ... and pts_i - pts_prev >= PTI, where checkpoint(pts_prev) = True and pts_prev = max (pts_0, pts_1, ..pts_i-1). If the test fails, the call results in a NOOP.

PHYSTIME_EXPLICIT: checkpoints are saved when the physics time equals or exceeds one of the (ordered) list of physics time values (in seconds) specified in the variable PHYSTIME_VALUES. Let [pt_0, pt_1, ..., pt_n] be the list of physics time values specified in the configuration parameter PHYSTIME_VALUES. Then checkpoint(pt) = True if pt >= pt_j, for some j in [0,n] and there is no other physics time pt_k, with pt > pt_k >= pt_j such that checkpoint(pt_k) = True. If the test fails, the call results in a NOOP.

The configuration parameter NUM_CHECKPOINT controls how many checkpoints to keep on disk. Checkpoints are deleted in a FIFO manner, based on their creation time. Possible values of NUM_CHECKPOINT are:

- NUM_CHECKPOINT = n, with n > 0 –> Keep the most recent n checkpoints
- NUM_CHECKPOINT = 0 –> No checkpoints are made/kept (except when *Force* = `True`)
- NUM_CHECKPOINT < 0 –> Keep ALL checkpoints

Checkpoints are saved in the directory `${SIM_ROOT}/restart`

cleanup()
Clean up any state from the services. Called by the terminate method in the base class for components.

create_simulation(*config_file*, *override*)
Create simulation

create_sub_workflow(*sub_name*, *config_file*, *override=None*, *input_dir=None*)
Create sub-workflow

create_task_pool(*task_pool_name*)
Create an empty pool of tasks with the name *task_pool_name*. Raise exception if duplicate name.

critical(*msg*, **args*)
Produce **critical** message in simulation log file. See `logging.critical()` for usage.

debug(*msg*, **args*)
Produce **debugging** message in simulation log file. See `logging.debug()` for usage.

error(*msg*, **args*)
Produce **error** message in simulation log file. See `logging.error()` for usage.

exception(*msg*, **args*)
Produce **exception** message in simulation log file. See `logging.exception()` for usage.

get_config_param(*param*, *silent=False*)
Return the value of the configuration parameter `param`. Raise exception if not found and silent is False.

Parameters

- **param** (*str*) – The parameter requested from simulation config
- **silent** (*bool*) – If True and parameter isn't found then exception is not raised, default False

Returns dictionary of given parameter from configuration

Return type dict

get_finished_tasks(*task_pool_name*)
Return dictionary of finished tasks and return values in task pool *task_pool_name*. Raise exception if no active or finished tasks.

get_port(*port_name*)

Parameters **port_name** (*str*) – port name

Returns Return a reference to the component implementing port *port_name*.

Return type *ipsframework.componentRegistry.ComponentID*

get_restart_files(*restart_root*, *timeStamp*, *file_list*)
Copy files needed for component restart from the restart directory:

```
<restart_root>/restart/<timeStamp>/components/$CLASS_${SUB_CLASS}_$NAME_${SEQ_
↪NUM}
```

to the component's work directory.

Copying errors are not fatal (exception raised).

get_time_loop()
Return the list of times as specified in the configuration file.

Returns list of times

Return type list of float

get_working_dir()
Return the working directory of the calling component.

The structure of the working directory is defined using the configuration parameters *CLASS*, *SUB_CLASS*, and *NAME* of the component configuration section. The structure of the working directory is:

```
${SIM_ROOT}/work/$CLASS_${SUB_CLASS}_$NAME_<instance_num>
```

Returns working directory

Return type str

info(*msg*, **args*)
Produce **informational** message in simulation log file. See `logging.info()` for usage.

kill_all_tasks()
Kill all tasks associated with this component.

kill_task(*task_id*)
Kill launched task *task_id*. Return if successful. Raises exceptions if the task or process cannot be found or killed successfully.

Parameters **task_id** (*int*) – task ID

Returns if successfully killed

Return type bool

launch_task(*nproc*, *working_dir*, *binary*, **args*, ***keywords*)
Launch *binary* in *working_dir* on *nproc* processes. **args* are any arguments to be passed to the binary on the command line. ***keywords* are any keyword arguments used by the framework to manage how the binary is launched. Keywords may be the following:

- *task_ppn* : the processes per node value for this task
- *task_cpp* : the cores per process, only used when `MPIRUN=srun` commands
- *task_gpp* : the gpus per process, only used when `MPIRUN=srun` commands
- ***omp*** [If `True` the task will be launch with the correct OpenMP environment] variables set, only used when `MPIRUN=srun`
- *block* : specifies that this task will block (or raise an exception) if not enough resources are available to run immediately. If `True`, the task will be retried until it runs. If `False`, an exception is raised indicating that there are not enough resources, but it is possible to eventually run. (default = `True`)
- *tag* : identifier for the portal. May be used to group related tasks.
- *logfile* : file name for `stdout` (and `stderr`) to be redirected to for this task. By default `stderr` is redirected to `stdout`, and `stdout` is not redirected.
- *whole_nodes* : if `True`, the task will be given exclusive access to any nodes it is assigned. If `False`, the task may be assigned nodes that other tasks are using or may use.

- *whole_sockets* : if `True`, the task will be given exclusive access to any sockets of nodes it is assigned. If `False`, the task may be assigned sockets that other tasks are using or may use.
- *launch_cmd_extra_args* : extra command arguments added the the MPIRUN command

Return *task_id* if successful. May raise exceptions related to opening the logfile, being unable to obtain enough resources to launch the task (*InsufficientResourcesException*), bad task launch request (*ResourceRequestMismatchException*, *BadResourceRequestException*) or problems executing the command. These exceptions may be used to retry launching the task as appropriate.

Note: This is a nonblocking function, users must use a version of *ServicesProxy.wait_task()* to get result.

Parameters

- **nproc** (*int*) – number of processes
- **working_dir** (*str*) – change to this directory before launching task
- **binary** (*str*) – command to execute, can include arguments or can be pass in with **args*

Returns task_id (PID)

Return type int

launch_task_pool(*task_pool_name*, *launch_interval=0.0*)

Construct messages to task manager to launch each task in task pool. Used by *TaskPool* to launch tasks in a task_pool.

Parameters

- **task_pool_name** (*str*) – name of task pool
- **launch_internal** (*float*) – time to wait between launching tasks, default 0.0

Returns activate task, dictionary mapping task_name to task_id

Return type dict

log(*msg*, **args*)

Wrapper for *ServicesProxy.info()*.

merge_current_state(*partial_state_file*, *logfile=None*, *merge_binary=None*)

Merge partial plasma state with global state. Partial plasma state contains only the values that the component contributes to the simulation. Raise exceptions on bad merge. Optional *logfile* will capture `stdout` from merge. Optional *merge_binary* specifies path to executable code to do the merge (default value : "update_state")

process_events()

Poll for events on subscribed topics.

publish(*topicName*, *eventName*, *eventBody*)

Publish event consisting of *eventName* and *eventBody* to topic *topicName* to the IPS event service.

remove_task_pool(*task_pool_name*)

Kill all running tasks, clean up all finished tasks, and delete task pool.

save_restart_files(*timeStamp*, *file_list*)

Copy files needed for component restart to the restart directory:

```
${SIM_ROOT}/restart/$timestamp/components/$CLASS_${SUB_CLASS}_$NAME
```

Copying errors are not fatal (exception raised).

send_portal_event(*event_type='COMPONENT_EVENT'*, *event_comment=''*, *event_time=None*, *elapsed_time=None*)
Send event to web portal.

setMonitorURL(*url=''*)
Send event to portal setting the URL where the monitor component will put data.

set_config_param(*param*, *value*, *target_sim_name=None*)
Set configuration parameter *param* to *value*. Raise exceptions if the parameter cannot be changed or if there are problems setting the value. This tell the framework to call *ipsframework.configurationManager.ConfigurationManager.set_config_parameter()* to change the parameter.

Parameters

- **param** (*str*) – The parameter requested from simulation config
- **value** – The value to set the parameter

Returns return value from setting parameter

stage_input_files(*input_file_list*)
Copy component input files to the component working directory (as obtained via a call to *ServicesProxy.get_working_dir()*). Input files are assumed to be originally located in the directory variable *INPUT_DIR* in the component configuration section.

File are copied using *ipsframework.ipsutil.copyFiles()*.

Parameters **input_file_list** (*str or Iterable of str*) – input files can space separated string or iterable of strings

stage_output_files(*timeStamp*, *file_list*, *keep_old_files=True*, *save_plasma_state=True*)
Copy associated component output files (from the working directory) to the component simulation results directory. Output files are prefixed with the configuration parameter *OUTPUT_PREFIX*. The simulation results directory has the format:

```
${SIM_ROOT}/simulation_results/<timeStamp>/components/$CLASS_${SUB_CLASS}_$NAME_
↪${SEQ_NUM}
```

Additionally, plasma state files are archived for debugging purposes:

```
${SIM_ROOT}/history/plasma_state/<file_name>_$CLASS_${SUB_CLASS}_$NAME_
↪<timeStamp>
```

Copying errors are not fatal (exception raised).

stage_state(*state_files=None*)
Copy current state to work directory.

stage_subflow_output_files(*subflow_name='ALL'*)
Gather outputs from sub-workflows. Sub-workflow output is defined to be the output files from its DRIVER component as they exist in the sub-workflow driver's work area at the end of the sub-simulation. If subflow_name != 'ALL' then get output from only that sub-flow

submit_tasks(*task_pool_name*, *block=True*, *use_dask=False*, *dask_nodes=1*, *dask_ppw=None*, *launch_interval=0.0*, *use_shifter=False*, *shifter_args=None*, *dask_worker_plugin=None*, *dask_worker_per_gpu=False*)
Launch all unfinished tasks in task pool *task_pool_name*. If *block* is `True`, return when all tasks have been

launched. If *block* is `False`, return when all tasks that can be launched immediately have been launched. Return number of tasks submitted.

Optionally, dask can be used to schedule and run the task pool.

subscribe(*topicName*, *callback*)
Subscribe to topic *topicName* on the IPS event service and register *callback* as the method to be invoked when an event is published to that topic.

unsubscribe(*topicName*)
Remove subscription to topic *topicName*.

update_state(*state_files=None*)
Copy local (updated) state to global state. If no state files are specified, component configuration specification is used. Raise exceptions upon copy.

update_time_stamp(*new_time_stamp=- 1*)
Update time stamp on portal.

wait_call(*call_id*, *block=True*)
If *block* is `True`, return when the call has completed with the return code from the call. If *block* is `False`, raise *IncompleteCallException* if the call has not completed, and the return value is it has.

Parameters **call_id** (*int*) – call ID

Returns service response message arguments

wait_call_list(*call_id_list*, *block=True*)
Check the status of each of the call in *call_id_list*. If *block* is `True`, return when *all* calls are finished. If *block* is `False`, raise *IncompleteCallException* if *any* of the calls have not completed, otherwise return. The return value is a dictionary of *call_ids* and return values.

Parameters **call_id_list** (*list of int*) – list of call ID's

Returns dict of call_id and return value

Return type dict

wait_task(*task_id*, *timeout=- 1*, *delay=1*)
Check the status of task *task_id*. Return the return value of the task when finished successfully. Raise exceptions if the task is not found, or if there are problems finalizing the task.

Parameters

- **task_id** (*int*) – task ID (PID)
- **timeout** (*float*) – maximum time to wait for task to finish, default -1 (no timeout)
- **delay** (*float*) – time to wait before checking if task has timed-out

Returns return value of task

wait_task_nonblocking(*task_id*)
Check the status of task *task_id*. If it has finished, the return value is populated with the actual value, otherwise `None` is returned. A *KeyError* exception may be raised if the task is not found.

Parameters **task_id** (*int*) – task ID (PID)

Returns return value of task if finished else None

wait_tasklist(*task_id_list*, *block=True*)
Check the status of a list of tasks. If `block` is `True`, return a dictionary of return values when *all* tasks have completed. If `block` is `False`, return a dictionary containing entries for each *completed* task. Note that the dictionary may be empty. Raise `KeyError` exception if `task_id` not found.

Parameters

- **task_id_list** (*list of int*) – list of task_id's (PID's) to wait until completed
- **block** (*bool*) – if to wait until all task finish

Returns dict of task_id and return value

Return type dict

warning(*msg*, **args*)
Produce **warning** message in simulation log file. See `logging.warning()` for usage.

class ipsframework.services.**Task**(*task_name*, *nproc*, *working_dir*, *binary*, **args*, ***keywords*)
Container for task information:

Parameters

- **name** (*str*) – task name
- **nproc** (*int*) – number of processes the task needs
- **working_dir** (*str*) – location to launch task from
- **binary** (*str*) – full path to executable to launch
- ***args** – arguments for *binary*
- ****keywords** – keyword arguments for launching the task. See *ServicesProxy.launch_task()* for details.

class ipsframework.services.**TaskPool**(*name*, *services*)
Class to contain and manage a pool of tasks.

add_task(*task_name*, *nproc*, *working_dir*, *binary*, **args*, ***keywords*)
Create *Task* object and add to *queued_tasks* of the task pool. Raise exception if task name already exists in task pool.

Parameters

- **task_name** (*str*) – unique task name
- **nproc** (*int*) – number of process to run task with
- **working_dir** (*str*) – change to this directory before launching task
- **binary** (*str*) – full path to executable to launch

get_dask_finished_tasks_status()
Return a dictionary of exit status values for all dask tasks that have finished since the last time finished tasks were polled.

Returns dict mapping task name to exit status

Return type dict

get_finished_tasks_status()
Return a dictionary of exit status values for all tasks that have finished since the last time finished tasks were polled.

Returns dict mapping task name to exit status

Return type dict

submit_dask_tasks(*block=True*, *dask_nodes=1*, *dask_ppw=None*, *use_shifter=False*, *shifter_args=None*, *dask_worker_plugin=None*, *dask_worker_per_gpu=False*)
Launch tasks in *queued_tasks* using dask.

One dask worker will be started for each node unless dask_worker_per_gpu is True where one dask worker will be started for every GPU. So dask_node times GPUS_PER_NODE workers will be started.

Parameters

- **block** (*bool*) – Unused, this will always return after tasks are submitted
- **dask_nodes** (*int*) – Number of task nodes, default 1
- **dask_ppw** (*int*) – Number of processes per dask worker, default is PROCS_PER_NODE
- **use_shifter** (*bool*) – Option to launch dask scheduler and workers in shifter container
- **dask_worker_plugin** (*distributed.diagnostics.plugin.WorkerPlugin*) – If provided this will be registered as a worker plugin with the dask client
- **dask_worker_per_gpu** (*bool*) – If true then a separate worker will be started for each GPU and binded to that GPU

submit_tasks(*block=True*, *use_dask=False*, *dask_nodes=1*, *dask_ppw=None*, *launch_interval=0.0*, *use_shifter=False*, *shifter_args=None*, *dask_worker_plugin=None*, *dask_worker_per_gpu=False*)

Launch tasks in *queued_tasks*. Finished tasks are handled before launching new ones. If *block* is `True`, the number of tasks submitted is returned after all tasks have been launched and completed. If *block* is `False` the number of tasks that can immediately be launched is returned.

If `use_dask==True` then the tasks are launched with *submit_dask_tasks()*. One dask worker will be started for each node unless dask_worker_per_gpu is True where one dask worker will be started for every GPU. So dask_node times GPUS_PER_NODE workers will be started.

Parameters

- **block** (*bool*) – If True then wait for task to complete, default True
- **use_dask** (*bool*) – If True then use dask to launch tasks, default False
- **dask_nodes** (*int*) – Number of task nodes, only used it `use_dask==True`
- **dask_ppw** (*int*) – Number of processes per dask worker, default is PROCS_PER_NODE, only used it `use_dask==True`
- **launch_internal** (*float*) – time to wait between launching tasks, default 0.0
- **use_shifter** (*bool*) – Option to launch dask scheduler and workers in shifter container
- **shifter_args** (*str*) – Optional arguments added to shifter when launching dask scheduler and workers
- **dask_worker_plugin** (*distributed.diagnostics.plugin.WorkerPlugin*) – If provided this will be registered as a worker plugin with the dask client
- **dask_worker_per_gpu** (*bool*) – If true then a separate worker will be started for each GPU and binded to that GPU

terminate_tasks()

Kill all active tasks, clear all queued, blocked and finished tasks.

ipsframework.services.**launch**(*binary*, *task_name*, *working_dir*, **args*, ***keywords*)

This is used by *TaskPool.submit_dask_tasks()* as the input to `dask.distributed.Client.submit()`.

5.10 Other Utilities

5.10.1 IPS Exceptions

exception ipsframework.ipsExceptions.**BadResourceRequestException**(*caller_id*, *tid*, *request*, *deficit*)
Exception raised by the resource manager when a component requests a quantity of resources that can never be satisfied during a get_allocation() call

exception ipsframework.ipsExceptions.**BlockedMessageException**(*msg*, *reason*)
Exception Raised by the any manager when a blocking service invocation is made, and the invocation result is not readily available.

exception ipsframework.ipsExceptions.**GPUResourceRequestMismatchException**(*caller_id*, *tid*, *ppn*, *gpp*, *max_gpp*)
Exception raised by the resource manager when it is possible to launch the requested number of GPUs per task

exception ipsframework.ipsExceptions.**IncompleteCallException**(*callID*)
Exception Raised by the taskManager when a nonblocking wait_call() method is invoked before the call has finished.

exception ipsframework.ipsExceptions.**InsufficientResourcesException**(*caller_id*, *tid*, *request*, *deficit*)
Exception Raised by the resource manager when not enough resources are available to satisfy an allocate() call

exception ipsframework.ipsExceptions.**InvalidResourceSettingsException**(*t*, *spn*, *cpn*)
Exception raised by the resource helper to indicate inconsistent resource settings.

exception ipsframework.ipsExceptions.**ResourceRequestMismatchException**(*caller_id*, *tid*, *nproc*, *ppn*, *max_procs*, *max_ppn*)
Exception raised by the resource manager when it is possible to launch the requested number of processes, but not on the requested number of processes per node.

exception ipsframework.ipsExceptions.**ResourceRequestUnequalPartitioningException**(*caller_id*, *tid*, *nproc*, *ppn*, *max_procs*, *max_ppn*)
Exception raised by the resource manager when it is possible to launch the requested number of processes, but the requested number of processes and processes per node will result in unequal partitioning of nodes.

5.10.2 IPS Utilities

ipsframework.ipsutil.**copyFiles**(*src_dir*, *src_file_list*, *target_dir*, *prefix=''*, *keep_old=False*)
Copy files in *src_file_list* from *src_dir* to *target_dir* with an optional prefix. If *keep_old* is `True`, existing files in *target_dir* will not be overridden, otherwise files can be clobbered (default). Wild-cards in file name specification are allowed.

ipsframework.ipsutil.**getTimeString**(*timeArg=None*)
Return a string representation of *timeArg*. *timeArg* is expected to be an appropriate object to be processed by `time.strftime()`. If *timeArg* is `None`, current time is used.

ipsframework.ipsutil.**which**(*program*, *alt_paths=None*)

class ipsframework.messages.**Message**(*sender_id*, *receiver_id*)
Base class for all IPS messages. **Should not be used in actual communication.**

FAILURE = 1

SUCCESS = 0

counter = 0

delimiter = ''

get_message_id()

identifier = 'MESSAGE'

class ipsframework.messages.**MethodInvokeMessage**(*sender_id*, *receiver_id*, *call_id*, *target_method*, **args*, ***keywords*)

Message used by components to invoke methods on other components.

- *sender_id*: component id of the sender
- *receiver_id*: component id of the receiver
- *call_id*: identifier of the call (generated by caller)
- *target_method*: method to be invoked on the receiver
- **args*: arguments to be passed to the *target_method*

counter = 0

delimiter = '|'

identifier = 'INVOKE'

class ipsframework.messages.**MethodResultMessage**(*sender_id*, *receiver_id*, *call_id*, *status*, **args*)

Message used to relay the return value after a method invocation.

- *sender_id*: component id of the sender (callee)
- *receiver_id*: component id of the receiver (caller)
- *call_id*: identifier of the call (generated by caller)
- *status*: either Message.SUCCESS or Message.FAILURE indicating the success of failure of the invocation.
- **args*: other information to be passed back to the caller.

counter = 0

delimiter = '|'

identifier = 'RESULT'

class ipsframework.messages.**ServiceRequestMessage**(*sender_id*, *receiver_id*, *target_comp_id*, *target_method*, **args*, ***keywords*)

Message used by components to request the result of a service action by one of the IPS managers.

- *sender_id*: component id of the sender
- *receiver_id*: component id of the receiver (framework)
- *target_comp_id*: component id of target component (typically framework)
- *target_method*: name of method to be invoked on component *target_comp_id*
- **args*: any number of arguments. These are specific to the target method.

counter = 0

delimiter = '|'

identifier = 'REQUEST'

class ipsframework.messages.**ServiceResponseMessage**(*sender_id*, *receiver_id*, *request_msg_id*, *status*, **args*)

Message used by managers to respond with the result of the service action to the calling component.

- *sender_id*: component id of the sender (framework)
- *receiver_id*: component id of the receiver (calling component)
- *request_msg_id*: id of request message this is a response to.
- *status*: either Message.SUCCESS or Message.FAILURE
- **args*: any number of arguments. These are specific to type of response.

counter = 0

delimiter = '|'

identifier = 'RESPONSE'

5.11 Framework Components

class ipsframework.portalBridge.**PortalBridge**(*services*, *config*)

Framework component to communicate with the SWIM web portal.

class SimulationData

Container for simulation data.

check_send_post_responses()

finalize(*timestamp=0.0*, ***keywords*)

Produce some default debugging information before the rest of the code is executed.

init(*timestamp=0.0*, ***keywords*)

Try to connect to the portal, subscribe to *_IPS_MONITOR* events and register callback `process_event()`.

init_simulation(*sim_name*, *sim_root*)

Create and send information about simulation *sim_name* living in *sim_root* so the portal can set up corresponding structures to manage data from the sim.

process_event(*topicName*, *theEvent*)

Process a single event *theEvent* on topic *topicName*.

send_event(*sim_data*, *event_data*)

Send contents of *event_data* and *sim_data* to portal.

send_mpo_data(*event_data*, *sim_data*)

step(*timestamp=0.0*, ***keywords*)

Poll for events.

ipsframework.portalBridge.**configure_mpo**()

ipsframework.portalBridge.**hash_file**(*file_name*)

Return the MD5 hash of a file :rtype: str :param file_name: Full path to file :return: MD5 of file_name

ipsframework.portalBridge.**send_post**(*conn*, *stop*, *url*)

---

class ipsframework.runspaceInitComponent.**runspaceInitComponent**(*services*, *config*)

Framework component to manage runspace initialization, container file management, and file staging for simulation and analysis runs.

init(*timestamp=0.0*, ***keywords*)
Creates base directory, copies IPS and FacetsComposer input files.

step(*timestamp=0.0*, ***keywords*)
Copies individual subcomponent input files into working subdirectories.

CHAPTER

SIX

INDEXES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W