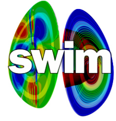


Oak Ridge National Laboratory

The SWIM Integrated Plasma Simulation Framework (IPS)

Wael R. Elwasif

elwasifwr@ornl.gov

And

The SWIM Project Team

Outline

- The IPS approach to coupled simulations.
- System architecture.
- Component interfaces and framework services.
- Simulation configuration management.
- Execution and monitoring.
- Adding new components to the IPS.

The IPS Approach to Coupled Simulation

- Part of c**SWIM**, the center for **S**imulation of RF **W**ave **I**nteractions with **M**agnetohydrodynamics.
- Primary directive:
  - ***Explore the targeted coupled physics interactions while constituent codes evolve independently, minimizing impact on long lived codes and other research/production use"***.
- Code re-factoring and/or rewriting ruled out.

Framework Design Guidelines

- Rapid, flexible coupling of ***existing*** simulation codes.
- Minimal (aka **NO**) change to underlying codes.
  - Not feasible to manage multiple branches.
- Simple coupling and integration protocol.
  - Maintainability, and ***debug-ability***.
- Integrated data management.
  - Simulation run as an experiment.
- Extensible simulation structure.
  - New physics components added as needed.

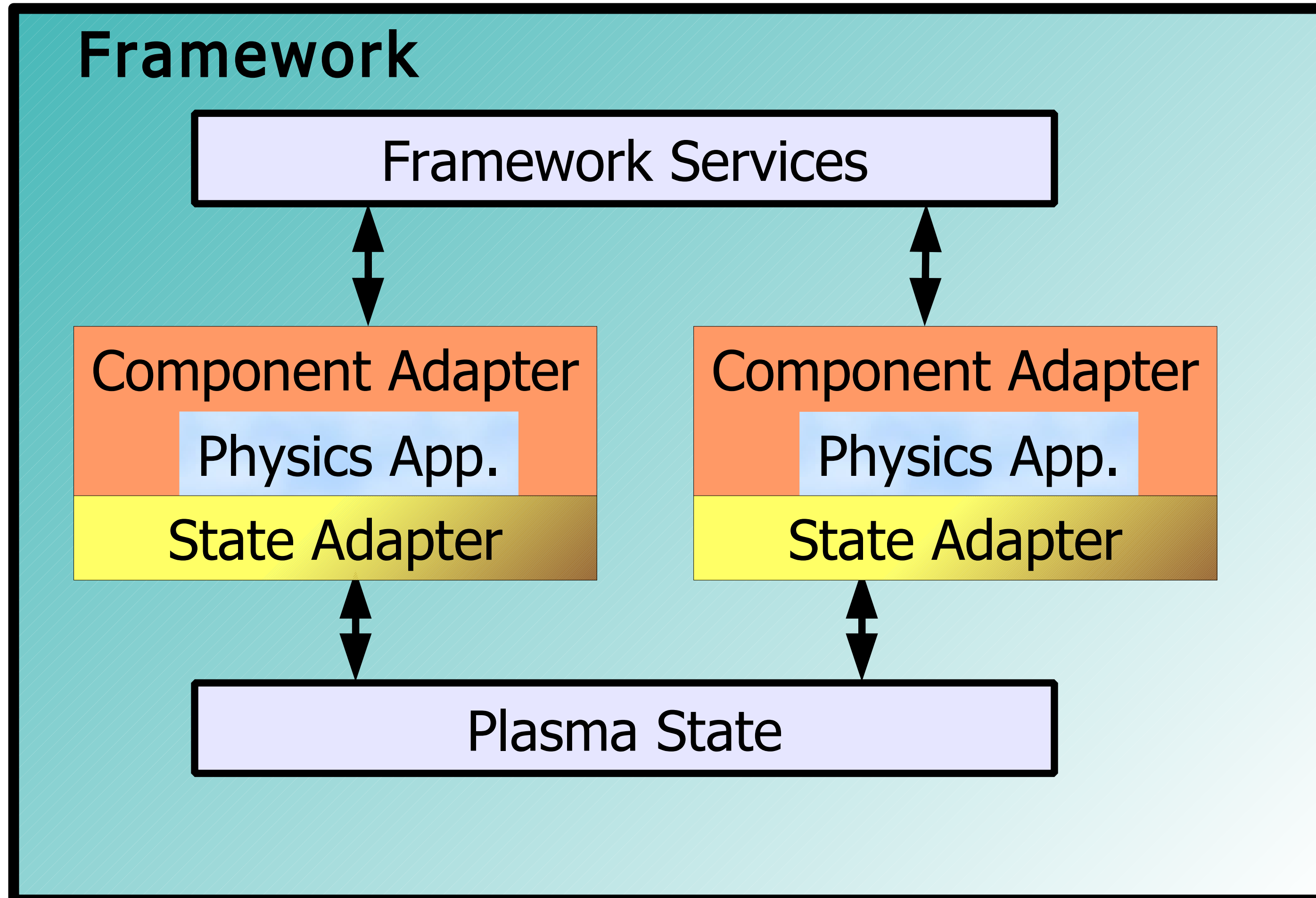
Major Design Features

- Component-based approach
  - ***Extensibility***, V&V, independent development.
- Common plasma state layer
  - Data repository.
  - Conduit for inter-component data exchange.
- File-Based data exchange
  - No change to underlying codes.
  - Simplify ***"unit testing"***

Major Design Features (2)

- Scripting Based Framework (Python)
  - RAD.
  - Adaptability, changeability, and flexibility.
- Simple component connectivity pattern
  - Driver/workers topology.
- Codes as components:
  - Focus on ***code-coupling*** vs ***physics-coupling*** as first step.
- Simple unified component interface
  - `init()`, `step()`, `finalize()`. *Too Simple??*

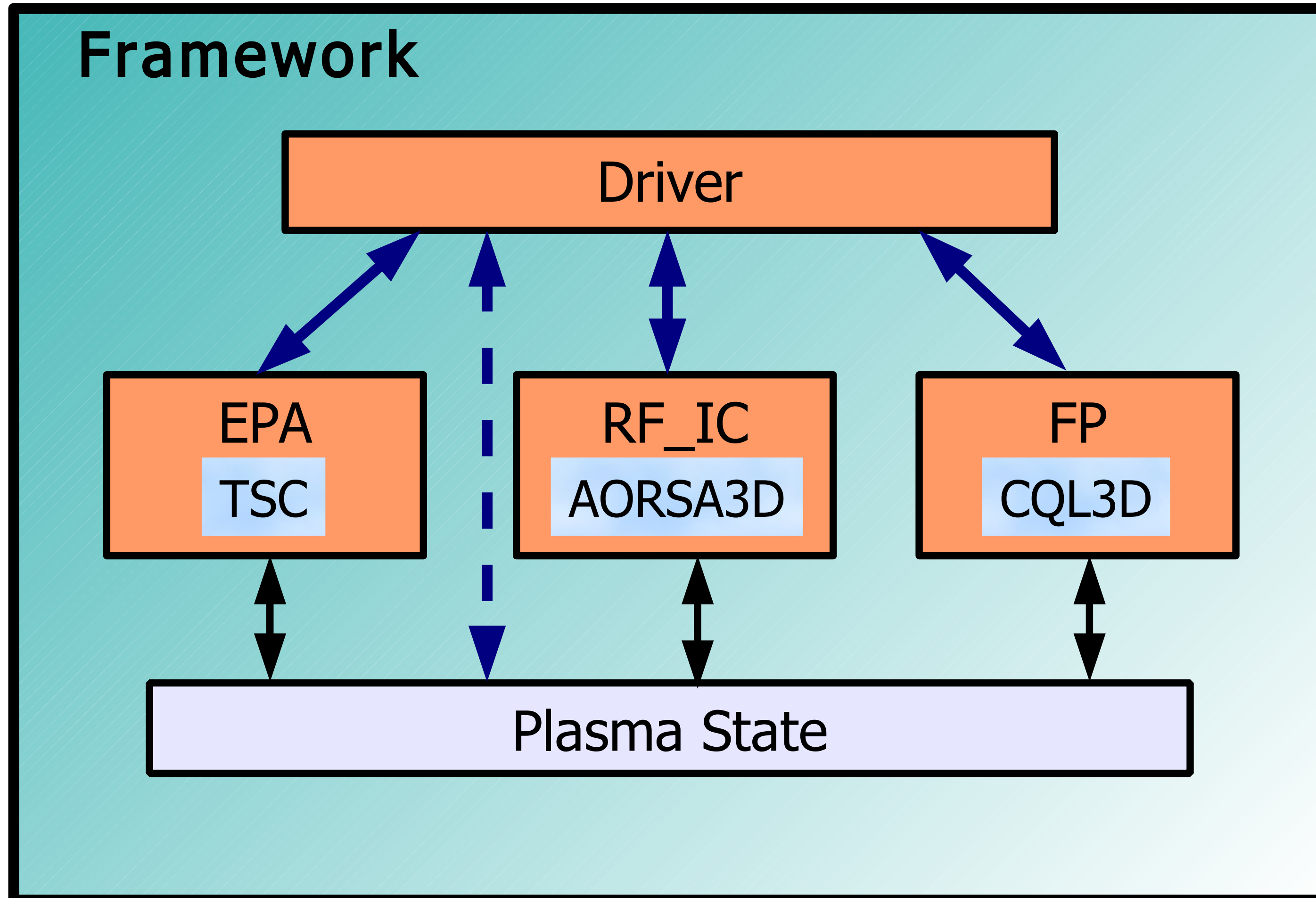
IPS Framework Layout



IPS Component Structure

- Component adapter:
  - Make a standalone app. into a component.
  - Utilize framework services and implement component interface.
- Underlying application:
  - Use ***unchanged*** in a coupled simulation.
- Plasma state adapter:
  - Map native app. data into common plasma state.
  - Receiver makes right.
  - Data definition and provenance??.

Sample IPS Application Structure



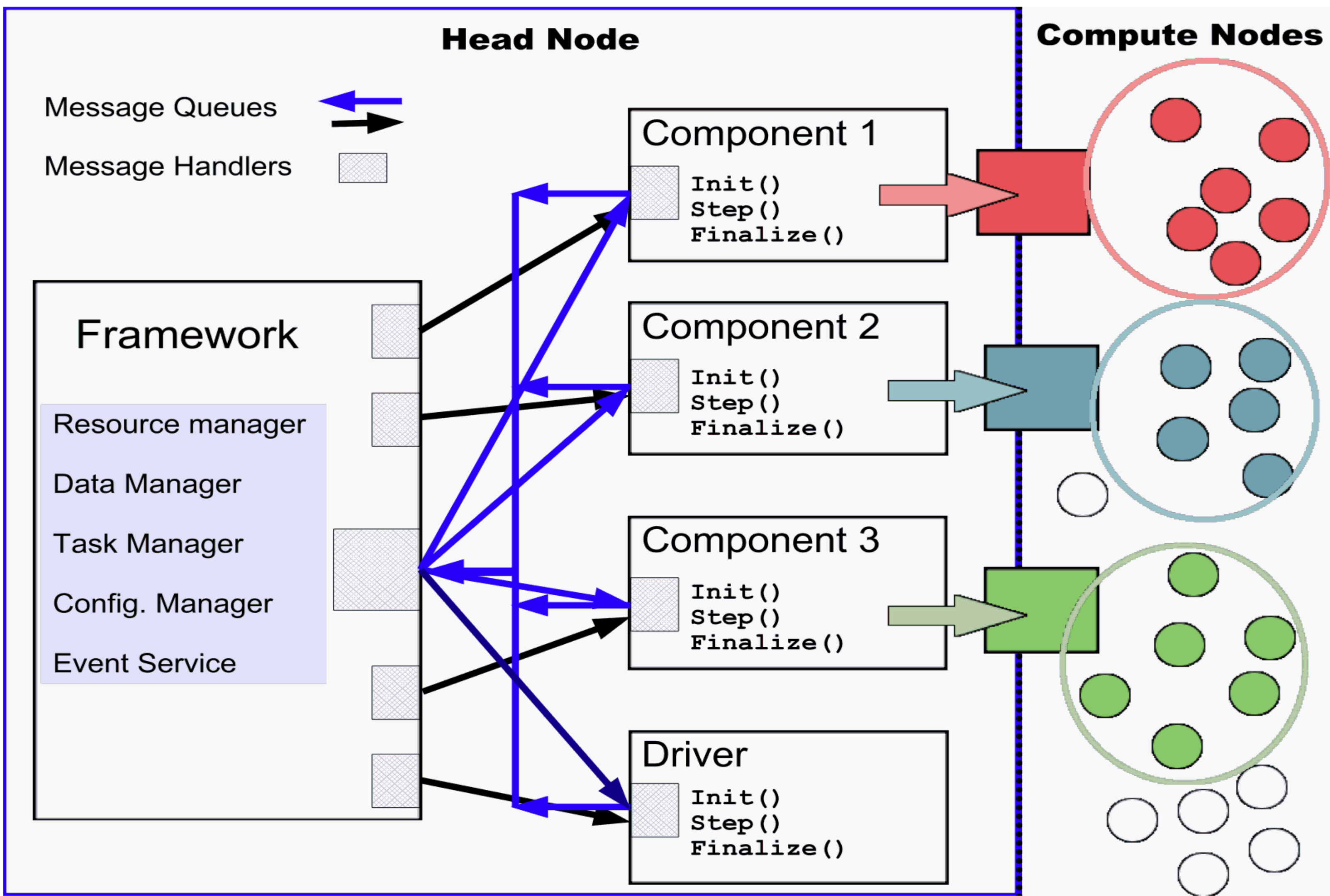
Framework Responsibilities

- *Configuration Management*
  - Simulation configuration.
  - Component instantiation and connection.
- *Task Management*
  - Mediate inter-component method invocation.
  - Manage execution of underlying applications.
- *Data Management*
  - Stage component input files.
  - Mediate shared access to plasma state files.
  - Archive component output files.

Framework Responsibilities (2)

- *Resource Management*
  - Manage access to computing resources (mainly compute nodes) for concurrent components.
- *Event Management*
  - Support asynchronous publish/subscribe model of data exchange in a running simulation.
- *Simulation Monitoring*
  - *Publish events to web-based SWIM portal.*

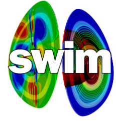
IPS Execution Environment



Invoking The IPS Framework

```
ips [--config=CONFIG_FILE_NAME]+ --platform=PLATFORM_FILE_NAME \
     --log=LOG_FILE_NAME [--debug] [--ftb]
```

- Support for multiple concurrent simulations, each with a separate simulation configuration file.
- Platform configuration file entries can be used and/or overridden in simulation config. files.
- Log file captures ***Framework*** logging output (not simulation logging).
- Debugging generates ***A LOT*** of entries in the log file.
- Integration with the Fault-tolerance Backplane Protocol using –**`ftb`** (Experimental)



IPS Configuration Files

- Using the syntax defined by Python's `ConfigObj` module.
  http://www.voidspace.org.uk/python/configobj.html
- Hierarchical sections using nested `[ ]` notation.
- Re-use of previously defined entries (similar to shell notation).
- Translates into a Python dictionary.
- *Re-use of platform configuration entries in the simulation configuration files enabled by the IPS framework.*

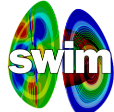
Platform Configuration File

```
HOST = franklin
#MPIRUN = eval
MPIRUN = aprun
PHYS_BIN_ROOT = /project/projectdirs/m876/phys-bin/phys/
        # Common location for underlying applications binaries.
PORTAL_URL = http://swim.gat.com:8080/monitor
RUNID_URL  = http://swim.gat.com:4040/runid.esp
        # URLs for communicating with the monitoring portal
DATA_ROOT = /project/projectdirs/m876/data/
        # Common location for simulation input data.
```

- Currently: franklin, jaguar, and viz/mhd at PPPL
- New entries added based on project needs.
- Components query for values as needed.
- Entries can be over-ridden in simulation configuration files.

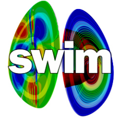
Simulation Configuration File

- Four major sections:
  - Global configuration options.
  - ***Ports*** configuration.
  - Components configuration.
  - Time loop specification.
- Ports
  - Map logical physics to concrete component implementations.
  - Allows for swapping *"equivalent"* components implementations without changing the driver.
  - Unique ports names, one component per port.



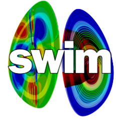
Configuration: Global Data

```
IPS_ROOT=/home/elwasif/ips/trunk # Root of IPS tree
SIM_NAME = AORSA_SIM             # Name of current simulation – Unique
SIM_ROOT = $IPS_ROOT/$SIM_NAME   # Simulation tree root – Unique
LOG_FILE  = $SIM_ROOT/$SIM_NAME.log  # Simulation log file – Unique
LOG_LEVEL = DEBUG                    # Default value: WARNING

RUN_ID = $SIM_NAME
OUTPUT_PREFIX =
CURRENT_STATE = ${RUN_ID}_ps.cdf
PRIOR_STATE = ${RUN_ID}_psp.cdf
CURRENT_EQDSK = ${RUN_ID}_ps.geq
PLASMA_STATE_FILES =$CURRENT_STATE $PRIOR_STATE $CURRENT_EQDSK
                      # What files constitute the plasma state
PLASMA_STATE_WORK_DIR = $SIM_ROOT/work/plasma_state
          # Where to put plasma state files as the simulation evolves
SIMULATION_MODE = NORMAL | RESTART             # Simulation mode
```



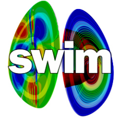
Configuration: Ports

```
[PORTS]
   NAMES = DRIVER INIT RF_IC EPA LINEAR_STABILITY FOKKER_PLANCK

   [[DRIVER]]                               # REQUIRED Port section
      IMPLEMENTATION = AORSA_CQL3D_DRIVER
                       # How is the simulation initialized
                       # (generate the very first state - if needed)
   [[INIT]]                                 # REQUIRED Port section - Warning
      IMPLEMENTATION = AORSA_CQL3D_INIT

   [[RF_IC]]
      IMPLEMENTATION = AORSA

   [[EPA]]
      IMPLEMENTATION = TSC

   [[FOKKER_PLANCK]]
      IMPLEMENTATION = CQL3D
```



Configuration: Component Implementation

```
[AORSA]
    CLASS = rf                                # Component categorization
    SUB_CLASS = ic
    NAME = aorsa                              # Component's Python class name.
    NPROC = 4                                 # Number of processors.
    BIN_PATH = $IPS_ROOT/bin                  # Where to look for application
                                              # Where to look for input files.
    INPUT_DIR = $DATA_ROOT/aorsa/ITER/CASE_10903
                                # List of input files
    INPUT_FILES = aorsa2d.in grfont.dat ZTABLE.TXT g096028.02650
                                # List of "important" output  files
    OUTPUT_FILES = out_swim out15 aorsa2d.ps aorsa2d.in

    SCRIPT = $BIN_PATH/rf_ic_aorsa.py   # Where to find the component
     . . .
     . . .
# Can/should add extra component-specific configuration entries here.
```

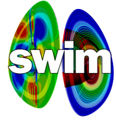


Configuration: Time Loop

```
#   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
   START = 3.5
   FINISH = 3.7
   NSTEP  = 2
   VALUES = 3.4 3.5 3.6 3.7
```

Component Details & Execution

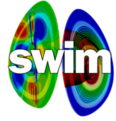
- Components are independent processes that communicate with the framework via messages.
- Each component instance executes in a separate work directory `$SIM_ROOT/work/$CLASS_$SUB_CLASS_$NAME_$INSTANCE#`.
- Direct access to component configuration variables (through the Python `self` variable).
  - `self.NPROC`, `self.INPUT_FILES`, `self.BIN_PATH`, ..etc
- Access to framework services through invocation on the `self.services` variable.
  - e.g. `self.services.get_port('RF')`



Oak Ridge National Laboratory

Framework Services Configuration Management

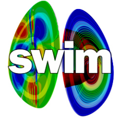
- `get_config_param(param)`
  - `sim_name = services.get_config_param('SIM_NAME')`
- `set_config_parameter(param, value)` `# Dynamic parameter set`
  - `services.set_config_parameter('FOO', 'FOO_VAL')`
- `get_port(port_name)`
  - `rf_comp_id = services.get_port('RF')`
- `get_time_loop()` `# Get a list of time values`
  - `time_list = services.get_time_loop()`
- `get_working_dir()`
  - `wdir = services.get_working_dir()`



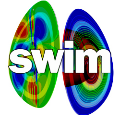
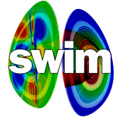
Oak Ridge National Laboratory

Framework Services: Task Management – Component Methods

- `call(component_id, method_name, *args)`
  - `ret_val = services.call(rf_comp_id, 'step', tlist[1])`
- `call_nonblocking(component_id, method_name, *args)`
  - `call_id = services.call_nonblocking(rf_comp_id, 'step', tlist[1])`
- `wait_call(call_id, block = True)`
  - `ret_val = services.wait_call(call_id)`
  - Raise **`IncompleteCallException`** if **block = False** and the call has yet to finish.
- `wait_call_list(call_id_list, block = True)`
  - `ret_val_dict = services.wait_call_list([call_id1, call_id2])`
    `retval_1 = ret_val_dict[call_id1]`
  - Undefined behavior with **`block = False`**.



Framework Services: Task Management – Task Execution

- `launch_task(nproc, working_dir, binary, *args, **keywords)`
  - ```
    cwd = services.get_working_dir()
    task = os.path.join(self.BIN_PATH, 'prepare_input')
    task_id = services.launch_task(1, cwd, task,
                                   logfile = 'prepare_input.log')
    ```
- `wait_task(task_id)`
  - `exit_val = services.wait_task(task_id)`
- `wait_task_nonblocking(task_id)`
  - `exit_val = services.wait_task_nonblocking(task_id)`
  - Return `None` if task has not finished.
- `kill_task(task_id)`
- `kill_all_tasks()`
    ```



Framework Services
Data management

- `stage_input_files(input_file_list)`
  - `stage_input_files(self.INPUT_FILES)`
- `stage_output_files(output_file_list)`
  - `stage_output_files(self.OUTPUT_FILES)`
- `stage_plasma_state()`
  - Copy from shared plasma directory to local working dir.
- `update_plasma_state(plasma_state_files=None)`
  - Copy from working directory to shared plasma dir
  - Default: update all files (can be overridden)
- `merge_current_plasma_state(partial_state_file, logfile=None)`
  - Update master current state, and copy it back to working dir.
  - Optional logfile argument to capture merge information.



OAK RIDGE National Laboratory

Framework Services
Event Management

- `publish(topicName, eventName, eventBody)`
  - ```
    event_data = {'key1': 'val1', 'key2': 345}
    services.publish('Topic1', 'New Event', event_data)
    ```
- `subscribe(topicName, callback)`
  - ```
    services.subscribe('Topic1', self.process_topic1)
    ```
  - ```
    def process_topic1(self, topicName, theEvent):
        event_body = theEvent.getBody()
        . . .
    ```
- `unsubscribe(topicName)`
- `process_events()`
  - Invoke callbacks on all outstanding events.
    ```

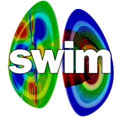
Framework Services : Logging

- `debug(*args)` `info(*args)`
  `warning(*args)` `error(*args)`
  `exception(*args)` `critical(*args)`
- `*args` conform to the Python logging module specification.
- Component-specific **`LOG_LEVEL`** overrides simulation-wide specification.
- Messages with severity that exceeds **`LOG_LEVEL`** appear in the simulation log file.
- `exception(*args)` can only be called from the **`except:`** part in a **`try: except:`** construct.

Plasma State Files

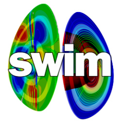
More Later

- "*THE*" PPPL Plasma state
  - NETCDF file with well-defined variables.
  - Accessible as a Fortran module (with supporting routines).
  - Extensible, auto-generated from a high level text description
  - Supports partial updating (for use by concurrent components).
- Other shared files:
  - Any other files accessed by more than one component.
  - No limit on number or type (performance issues for large files).



Creating IPS Components: The Code

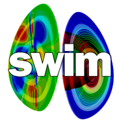
```
from  component import Component

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

  def init(self, timeStamp=0.0):
    return

  def step(self, timeStamp=0.0):
    return

  def finalize(self, timeStamp=0.0):
    return
```



Creating IPS Components: The Build System

- Add `Makefile` and `Makefile.include` to the component directory
  - Copy from other component directories, and adjust path to **`IPS_ROOT`**, target binaries, scripts, and dependencies.
  - Compiler and default linked libraries (plasma state) specified in top level `Makefile.config` – included in `Makefile.include`.
- Add entries to top level Makefile

```
HELLO_COMP_DIR=components/drivers/hello
HELLO_COMP=.HELLO_WORLD

COMPONENTS_DIRS = . .\
                 $(HELLO_COMP_DIR)

COMPONENTS = .. \
             $(HELLO_COMP)
```



Oak Ridge National Laboratory

Data Management
Simulation Tree Layout

- ***$SIM_ROOT***
  - Config File
  - simulation_setup — component scripts, input Files
  - simulation_log — Local portal events log
  - simulation_results — One dir per time step, symlinks in components dir
  - work — One dir per component instance.

Data Management: Issues

- Multiple runs can overlay their simulation results
  - Using **`OUTPUT_PREFIX`** configuration parameter.
- Plasma state files are archived in `simulation_results` whenever `stage_output_files()` is called.
  - Debug component effect on shared state.
- Use platform-wide **`DATA_ROOT`** to centralize input data storage.
- **`PHYS_BIN_ROOT`** (in platform configuration file) as canonical location for underlying applications.

Error Handling in IPS Components

- Errors represented as Python exceptions.

```
try:
    #services method invocation
except Exception:
    services.exception('Exception in call to services')
  raise
```

- Uncaught exception propagate across component processes to the framework:
  - Framework ends simulation with uncaught exceptions.

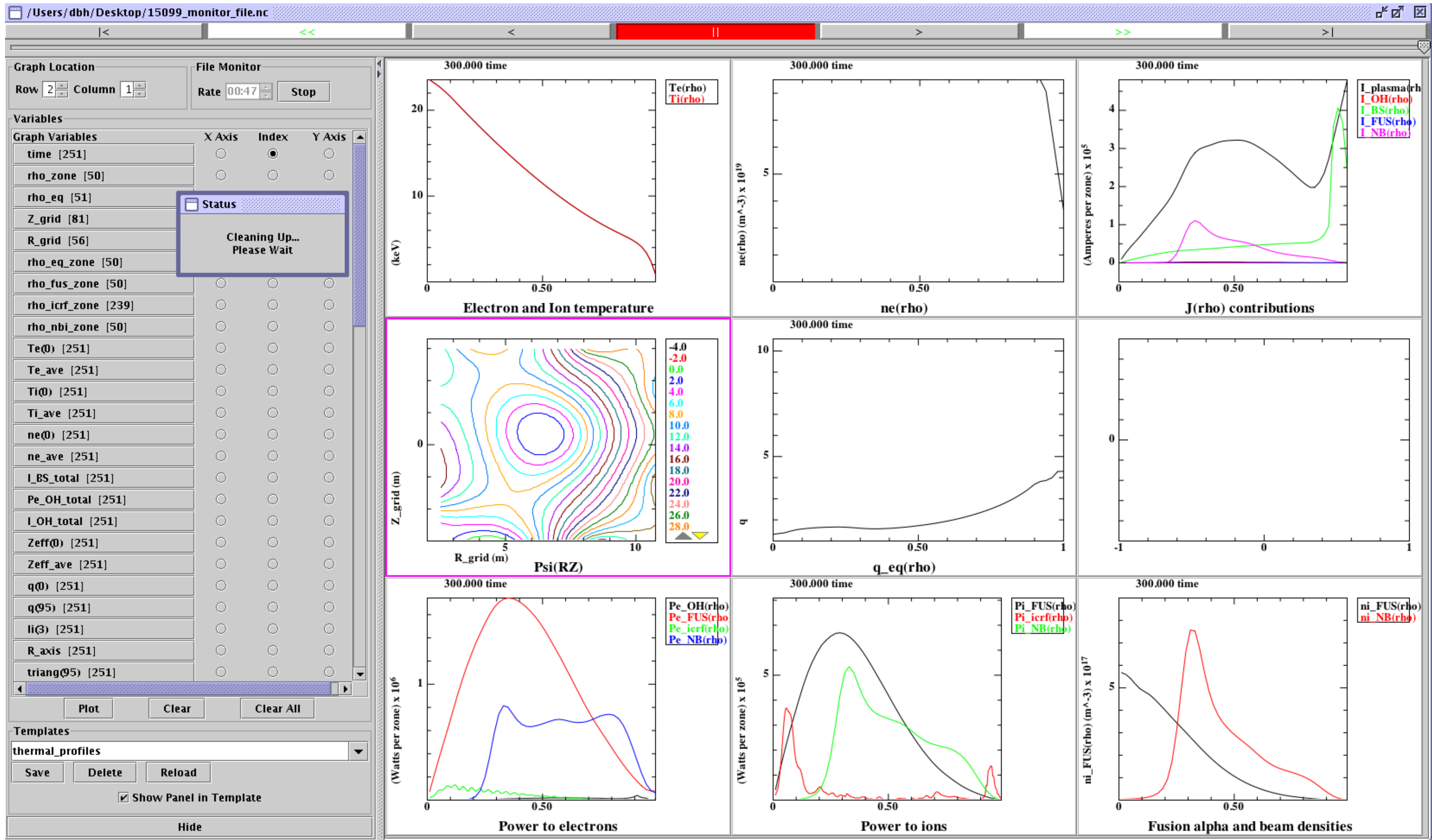
Simulation Execution

- Framework invoked in batch script, running on the *head node.*
- `create_batch_script.py` creates a template for PBS based systems, encoding some suggested practices.
- Multiple log files from batch submission:
  - Batch job stdout, stderr (one or two files)
  - Framework log file (from **`--log=`** command line option)
  - Simulation log file (one per simulation).

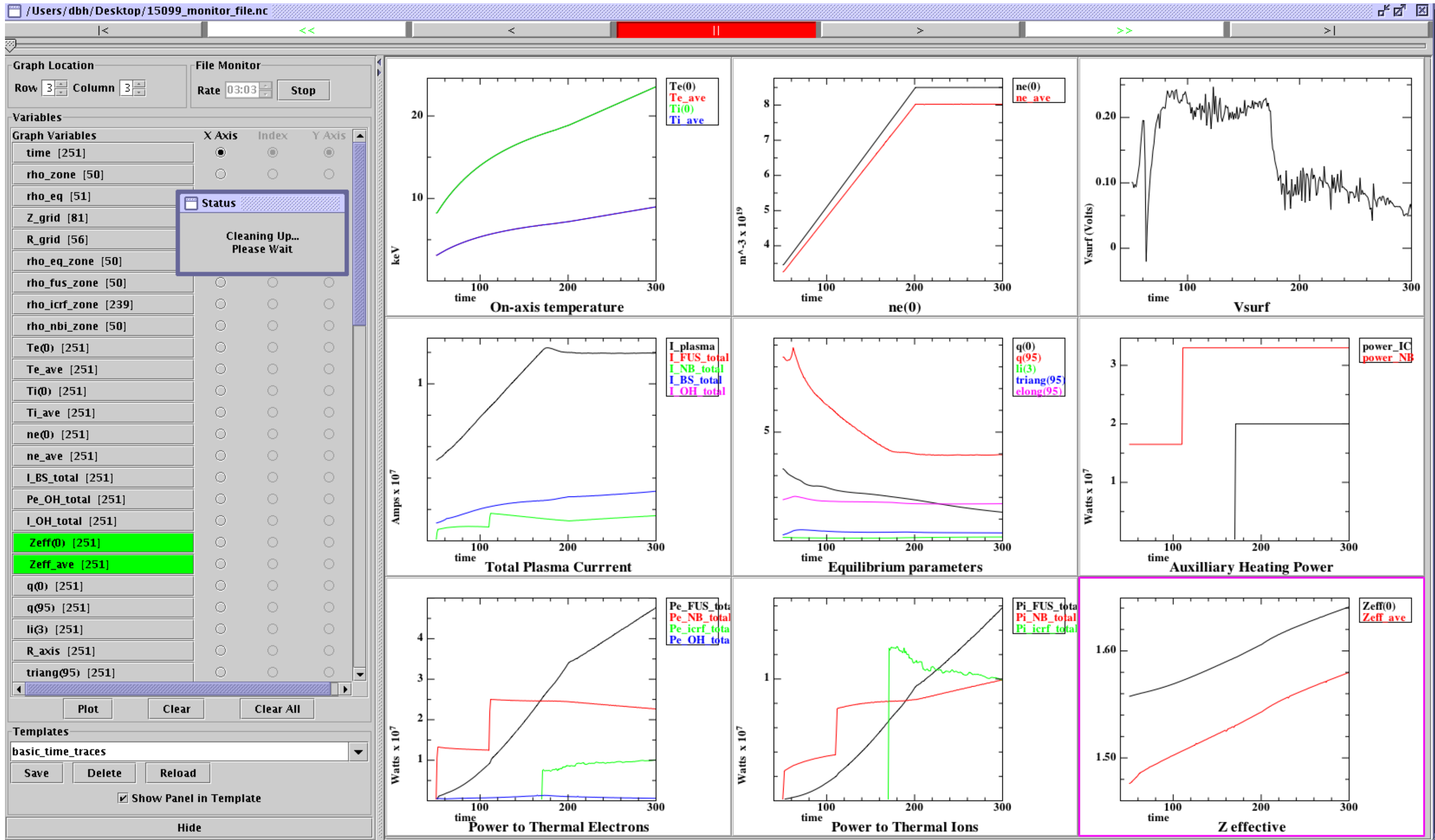
Simulation Monitoring

- Using a combination of events and a user-level component.
- The monitoring component extracts important data from the plasma state snapshots.
- A monitor file (NETCDF file with the unbounded time dimension) is placed in a Web accessible directory.
- A viewing template is used to configure the charts of interest.
- Elvis is used to render simulation progress
  - Used either as a browser plug-in or standalone desktop application.

Sample Display – Thermal Profiles



Sample Display – Basic Time Traces



SWIM Component Roster

- RF_IC
  - AORSA, TORIC
- Fokker-Planck
  - CQL3d
- Equilibrium & Profile Advance
  - TSC, JSOLVER
- Neutron-Beam:
  - NUBEAM
- Linear MHD:
  - PEST II, Balloon, Nova-k
- Miscellaneous:
  - Sim. Monitoring.
- In progress:
  - M3D, M3D-C1, NIMROD, GENRAY, TRXPL, ORBIT-RF