The Integrated Plasma Simulator: Framework for Loosely Coupled Codes

Wael R. Elwasif

elwasifwr@ornl.gov

And

The SWIM Project Team

Motivation & Environment

- Need for systemic coupling of various physics and codes
  - Prelude to FSP
- Established, long-lived codes used in various contexts.
  - Mainly stand-alone, occasionally two-way coupling.
- Different levels of parallelism.
  - From serial codes to massively parallel.
- No standard format for data exchange.
  - Several de-facto leaders in common use.
- No mandate to re-write major codes.
  - The goal is coupled physics, not code re-factoring.
- Codes ***WILL*** continue to evolve during project lifetime
  - Code forking to be avoided.

Framework Design Guidelines

- ***Flexibility***
  - Facilitate exploration of evolving (and changing) coupled physics
- ***Extensibility***
  - Not limited to pre-defined small set of codes.
  - Ability to add physics into the coupled model to extend its scope and/or fidelity
- ***Separation of concerns***
  - Minimally intrusive approach separating coupling infrastructure from core physics.
- ***Simplicity, Maintainability, and Debug-ability***
  - Light-weight coupling infrastructure.
- ***Support "off-HPC" aspects of running simulations***
  - Data management, monitoring, post-mortem analysis, . . .

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.

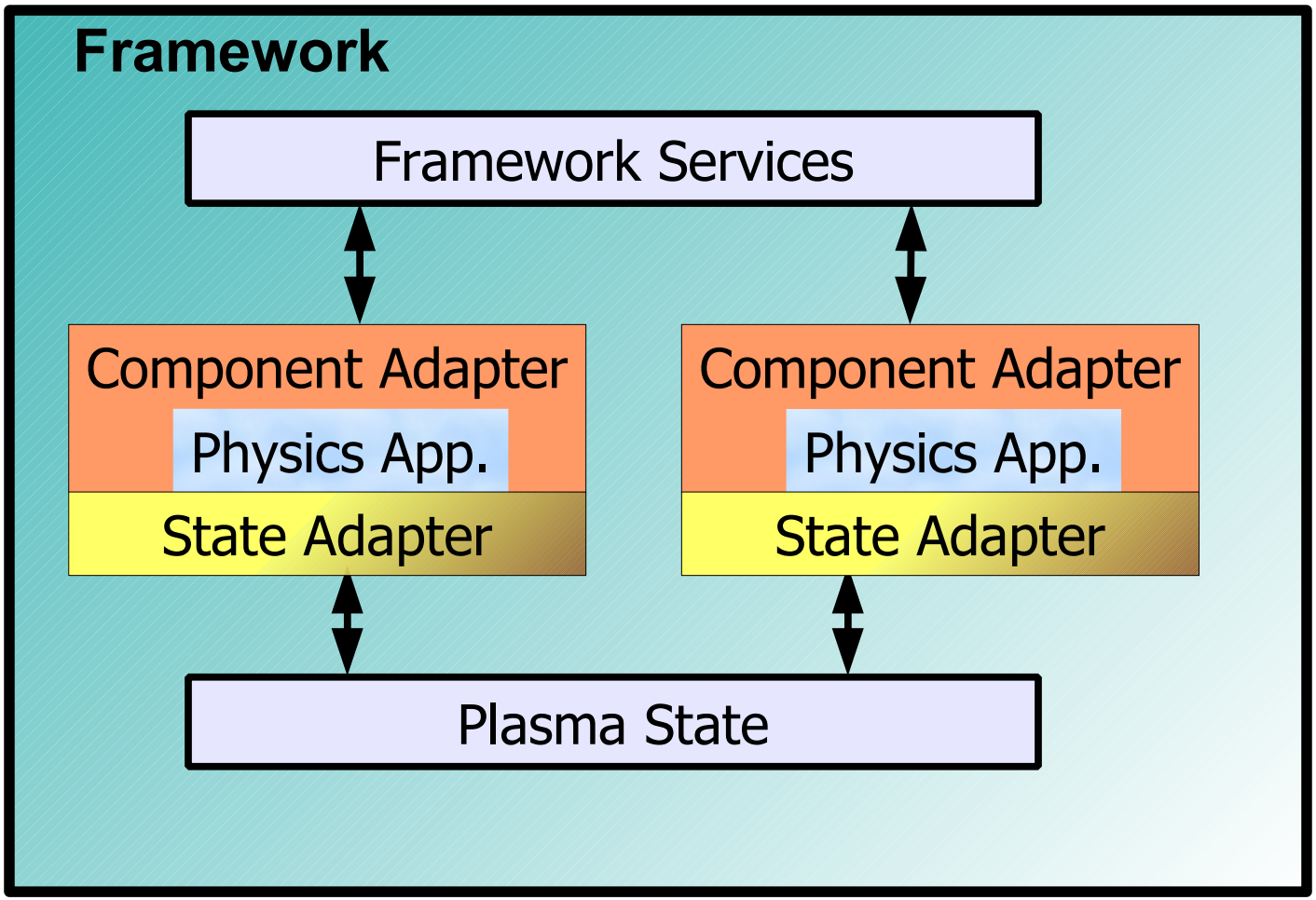
Framework Design Decisions

- Component-based approach
  - Inspired by the Common Component Architecture (CCA)
  - ***Extensibility***, V&V, independent development.
- Common plasma state layer
  - Standard file format for simulation data archival.
  - Conduit for inter-component data exchange.
- File-Based inter-component data exchange
  - No change to underlying codes.
  - Simplify ***"unit testing"***
  - ***The*** plasma state and/or any other files accessed by more than one component.

IPS Features

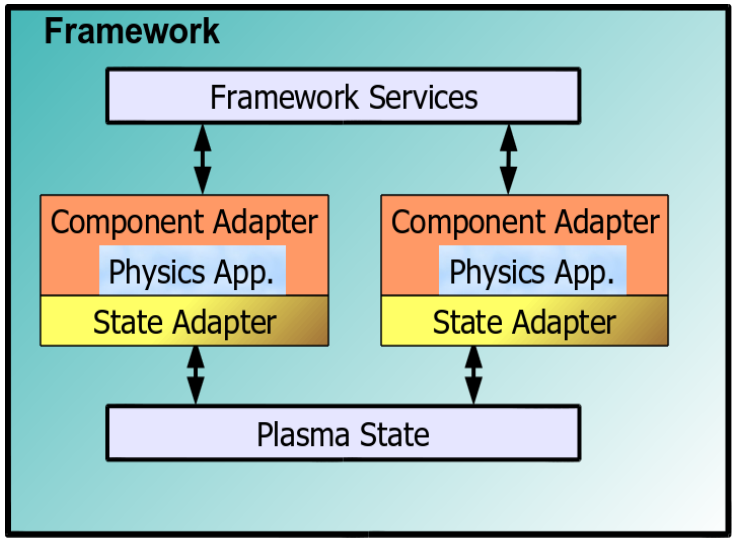
- Scripting Based Framework (Python)
  - Lightweight, rapid development
  - 22 files, ***6558 LOC*** (**767** blank, **1488** comment, **4303** code)
  - Ideal for "***off-HPC***" aspects of coupled simulations
  - Expressivity, adaptability, changeability, and flexibility.
- Simple component connectivity pattern
  - Driver/workers topology as "standard" connectivity pattern.
  - Other patterns possible as well
- Codes as components:
  - Focus on ***code-coupling*** vs ***physics-coupling*** as first step.
- Simple unified component interface
  - `init()`, `step()`, `finalize()` (plus `checkpoint()` `restart()`)

Framework Architectural Outline



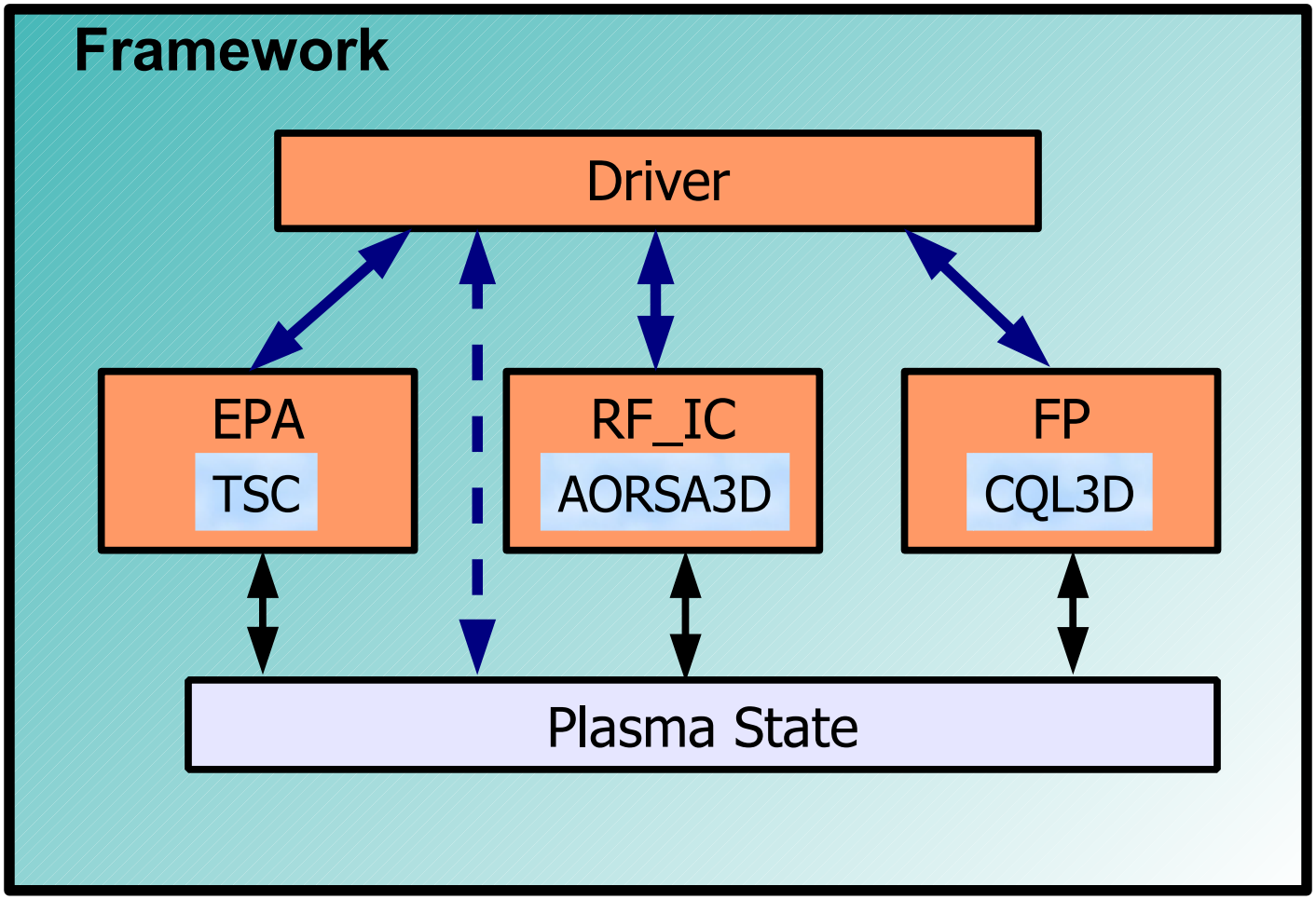
IPS Component Structure

- Component adapter:
  - Morph a standalone application into an compliant component.
  - Utilize framework services to implement component interface.
- Underlying application:
  - Use ***unchanged*** in a coupled simulation.

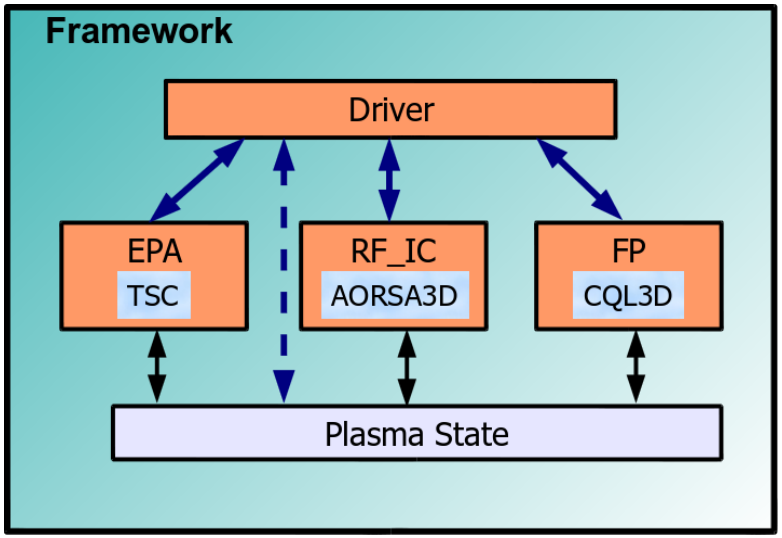


- Plasma state adapter:
  - Map pertinent native app. I/O data into common plasma state.
  - Receiver makes right (writer decides on format).
  - Data definition and provenance??.

Sample IPS Application Structure



Anatomy of an IPS Simulation



- Framework validates and instantiates all physics components
  - As well as any framework-attached components.
- A single ***DRIVER*** component specification is required per simulation.
  - Optional ***INIT*** component to initialize the plasma state – if needed.
- Framework invokes the driver component's public API in order (`init(), step(), finalize()`)
- Simulation ends when the driver's `finalize()` method returns (to the framework).
- The inter-component interactions within a simulation are initiated and controlled by the components themselves, *not the framework*.

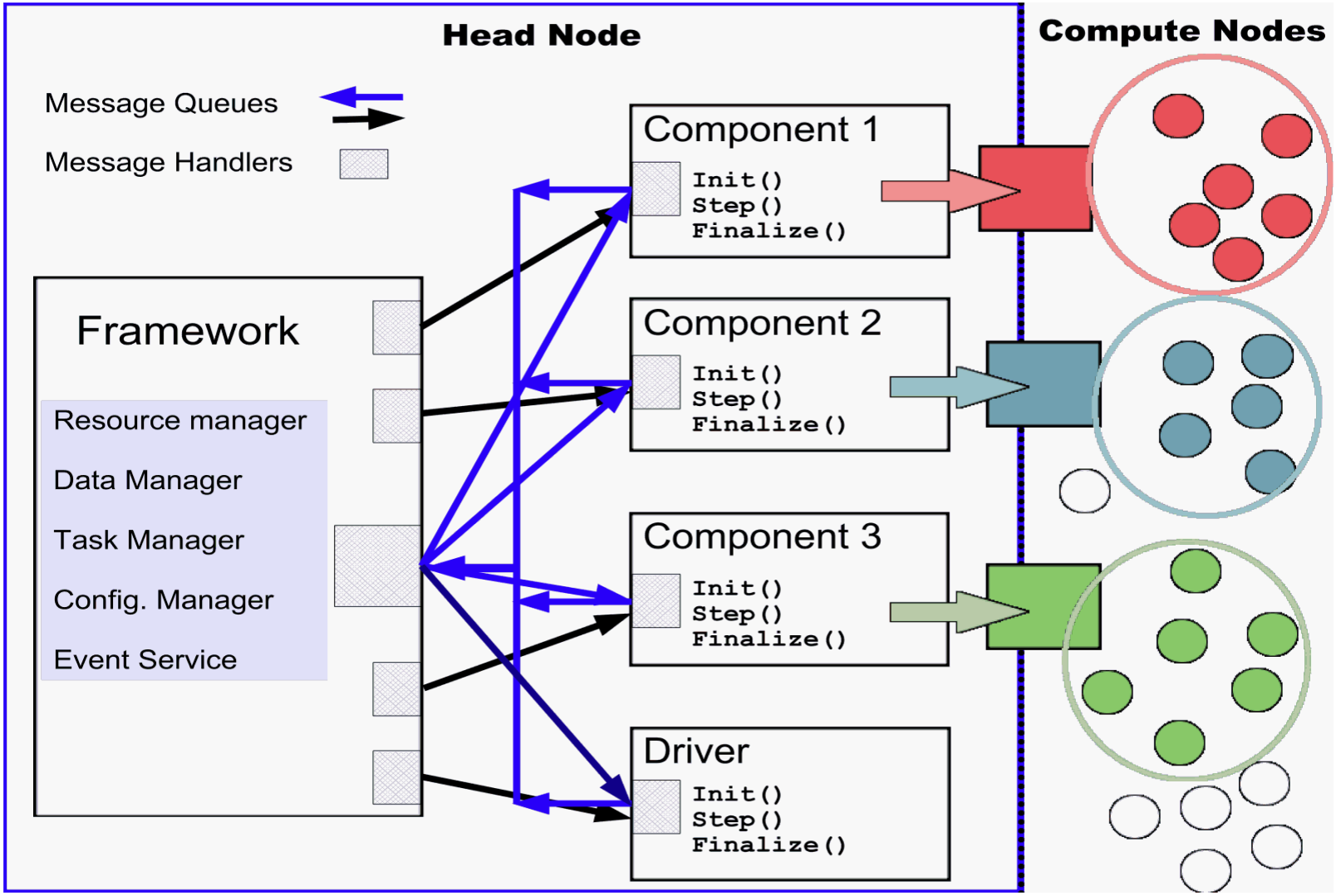
Framework Services

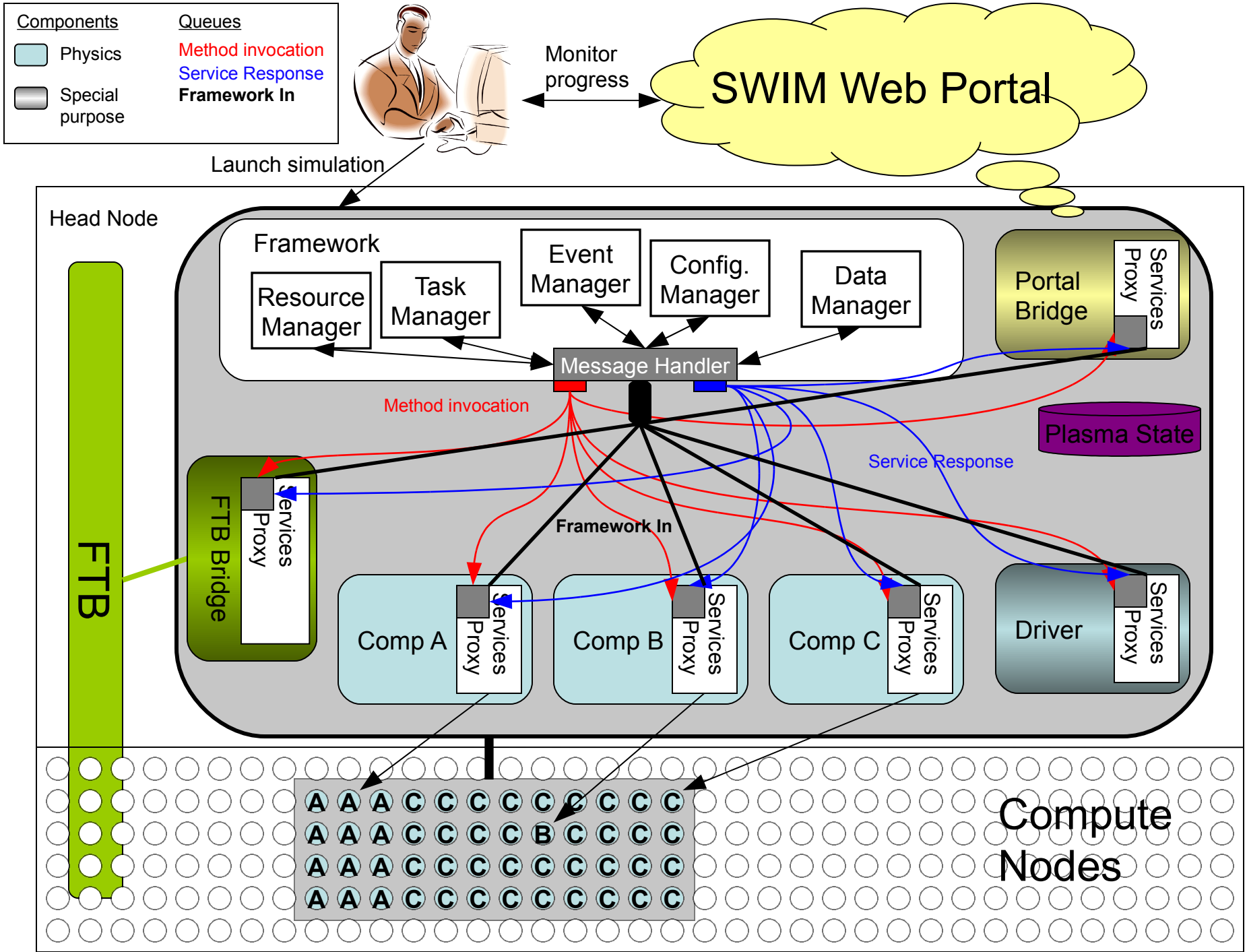
- *Configuration Management*
  - Simulation configuration.
  - Component instantiation and connection.
- *Task Management*
  - Mediate inter-component method invocation.
  - Manage execution of underlying applications.
- *Data Management*
  - Stage component input files.
  - Mediate shared access to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

Framework Servies (2)

- *Resource Management*
  - Manage access to computing resources (mainly compute nodes) for concurrent components.
- *Asynchronous 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




Components
Physics
Special purpose
Queues
Method invocation
Service Response
Framework In
Monitor progress
SWIM Web Portal
Launch simulation
Head Node
Framework
Resource Manager
Task Manager
Event Manager
Config. Manager
Data Manager
Message Handler
Portal Bridge
Services Proxy
Method invocation
Plasma State
Service Response
FTB
FTB Bridge
Framework In
Comp A
Comp B
Comp C
Driver
Compute Nodes

Invoking The IPS Framework

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

- Invocation typically done from within a batch script, but can also be done interactively.
- Support for multiple concurrent simulations, each with a separate simulation configuration file (more on that later).
- 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, hopper, jaguar, TechX, and stix @ PPPL
- New entries added based on project needs and users.
- Components query for values as needed.
- Entries can be over-ridden in simulation configuration files.

Simulation Configuration File

- Five major sections:
  - Global configuration options.
  - ***Ports*** configuration.
  - Components configuration.
  - Time loop specification.
  - Checkpoint schedule 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]]    # Optional Port section – Warning if absent
      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
```

Configuration: Checkpoint Schedule

```
#  MODE = WALLTIME_REGULAR | WALLTIME_EXPLICIT |
          PHYSTIME_REGULAR | PHYSTIME_EXPLICIT
#  Entries in the CHECKPOINT section vary depending on the value
#  of the MODE parameter

#  Example
[CHECKPOINT]
   MODE = WALLTIME_REGULAR
   NUM_CHECKPOINT = 4
   WALLTIME_INTERVAL = 1800
   PROTECT_FREQUENCY  = 10
```

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')`**

Framework Services
Configuration Management

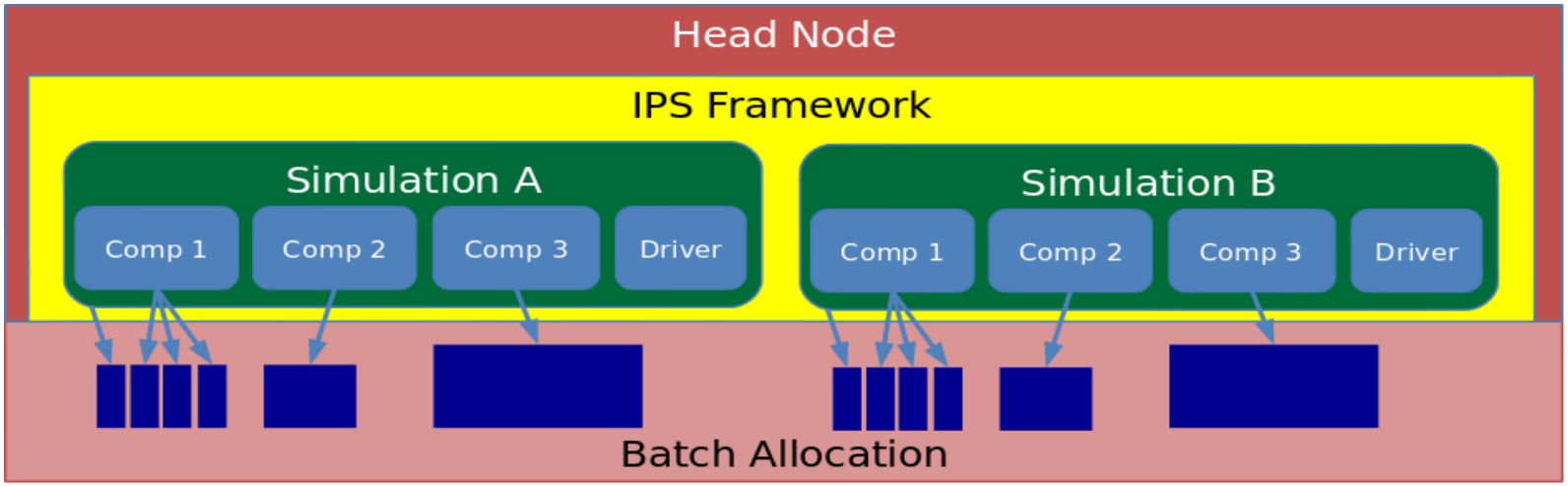
- Manage component access to global configuration data in the configuration file.
- Allow the dynamic creation and query of shared parameters at run time.
- Manage mapping between ports and implementation components
  - Mainly used by the driver

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()

Task Management
FOUR Levels of Concurrency

- Components launch parallel tasks (task = mpi application execution).
- A component can launch multiple tasks concurrently.
- Multiple component methods can be active concurrently.
  - From different components
- Multiple simulations can be managed by the framework concurrently
  - Sharing resources allocated to a single batch job submission.



Task Management : Parallel Tasks

- `launch_task(nproc, working_dir, binary, *args, **keywords)`
  - Optional Keywords:
    - `logfile = 'filename'` – Capture stdout & stderr from task
    - `task_ppn = 2` – configure cores per node to use for this task
- `wait_task(task_id)`

```
cmd = 'compute_parallel'
arg1 = 'dat.in'
arg2 = 'dat.out'
num_procs = N
task_dir = services.get_working_dir()
task_id = services.launch_task(num_procs, task_dir,
                  cmd, arg1, arg2, logfile = 'task.log')
exit_code = services.wait_task(task_id)
```

Task Management : Concurrent Tasks

- launch_task(nproc, working_dir, binary, *args, block = False)
- wait_task_nonblocking(task_id)

```
all_tasks = [task0, task1, task2, ..., taskn]
active_tasks = []
while True:
  try:
    new_task = all_tasks.pop()
  except IndexError:            # Empty list
    break
  try:
    task_id = services.launch_task(num_procs, task_dir, cmd, arg1, arg2,
              logfile = 'task.log', block = False)
  except InsufficientResourcesException:
    all_tasks.insert(0, new_task)
    exit_codes = services.wait_tasklist(active_tasks,
                                        block=True)
    active_tasks = []
  else:
    active_tasks.append(task_id)
if (len(active_tasks) > 0) :
  exit_codes = services.wait_tasklist(active_tasks, block=True)
```

Task Management : Task Pools

- Common Scenario:
  - Launch `N` tasks
  - Block pending all task termination
  - Task execution order immaterial.
- Examples:
  - Parallelize over flux surface.
  - Fine solver for Parareal parallel-in-time algorithm.
- Framework manages submission of tasks in the pool
- Non blocking mode allows for more user control
  - Allowing task addition to the pool after initial submission.

Task Pools : API & Example

- create_task_pool()
- add_task()
- submit_tasks()
- get_finished_tasks()

```
all_tasks = [task0, task1, task2, ..., taskn]
task_pool = services.create_task_pool('TASK POOL')
for i in range(len(all_tasks)):
  task_name = 'Task_%d' % (i)
  new_task = all_tasks[i]
  # Extract new task data here
  services.add_task(task_pool, task_name, num_procs,
                    task_dir,  cmd, arg1, arg2,
                    logfile = task_name + '.log')
services.submit_tasks('TASK POOL')
exit_status = services.get_finished_tasks('TASK POOL')
```

Blocking Component Methods

- `call(component_id, method_name, *args)`
  - `ret_val = services.call(rf_comp_id, 'step', t0)`
- Task Manager mediates calls and marshals arguments and return values.
- IPS implements *call-by-value.*

```
comp1_ref = services.get_port('RF')
comp2_ref = services.get_port('NB')
comp3_ref = service.get_port('EPA')
ret_val1 = services.call(comp1_ref, 'step', t)
ret_val2 = services.call(comp2_ref, 'step', t)
ret_val3 = services.call(comp3_ref, 'step', t)
```

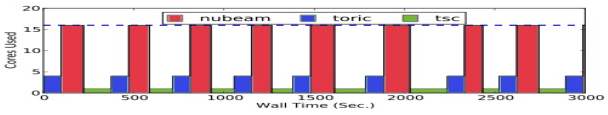
Concurrent Component Methods

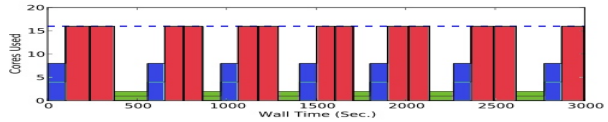
- call_nonblocking(component_id, method_name, *args)
- wait_call(call_id, block = True)
- wait_call_list(call_id_list, block = True)

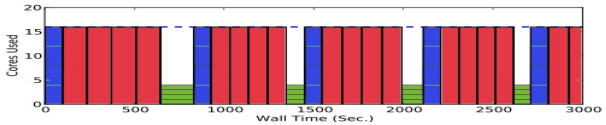
```
comp1_ref = services.get_port('RF')
comp2_ref = services.get_port('NB')
comp3_ref = service.get_port('EPA')
call_id1 = services.call_nonblocking(comp1_ref, 'step', t)
call_id2 = services.call_nonblocking(comp2_ref, 'step', t)
call_id3 = services.call_nonblocking(comp3_ref, 'step', t)
call_list = [call_id1, call_id2, call_id3]
retval_dict = services.wait_call_list(call_list)
```

Multiple Concurrent Simulations

- One framework, one batch job, N configuration files
- Interleaved concurrent tasks
  - Even when a simulation is sequential.
- Improve resource utilization and time to solution.

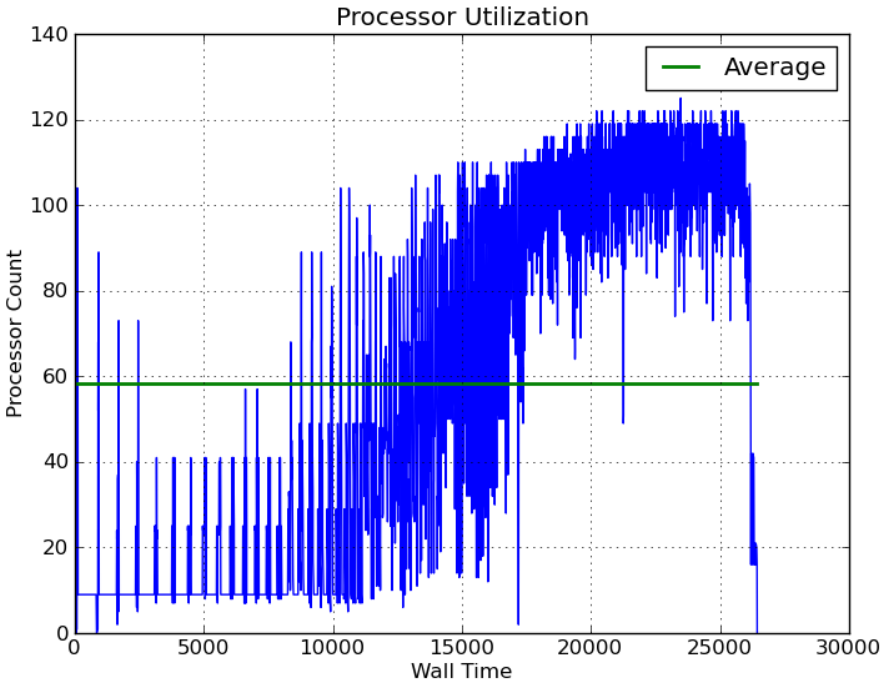


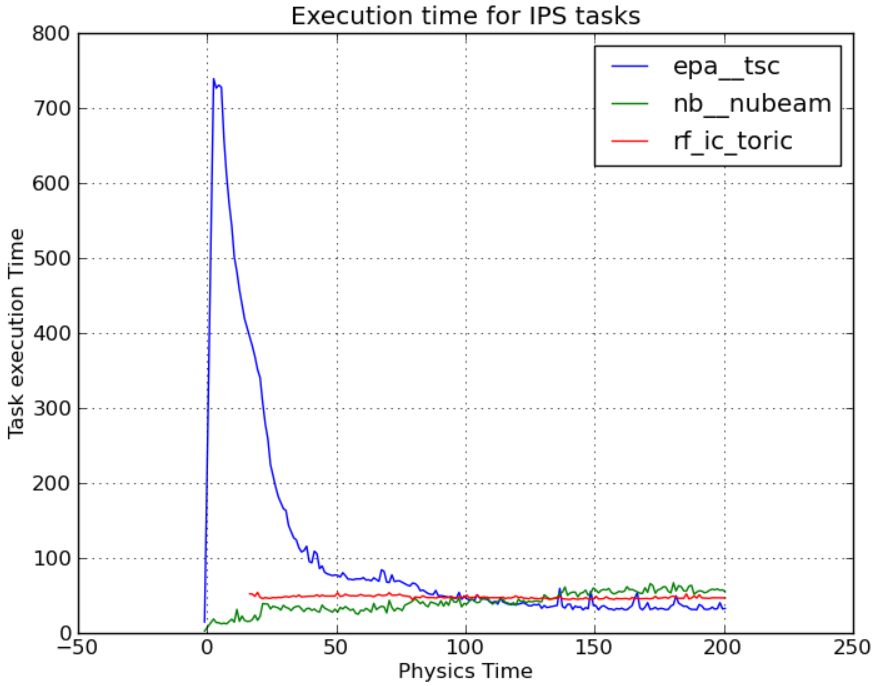




Case Study: ITER Pedestal Scan

- 9 Concurrent simulations
- TSC (serial) , TORIC (4 procs), NUBEAM (16 procs).





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.

Checkpoint/Restart

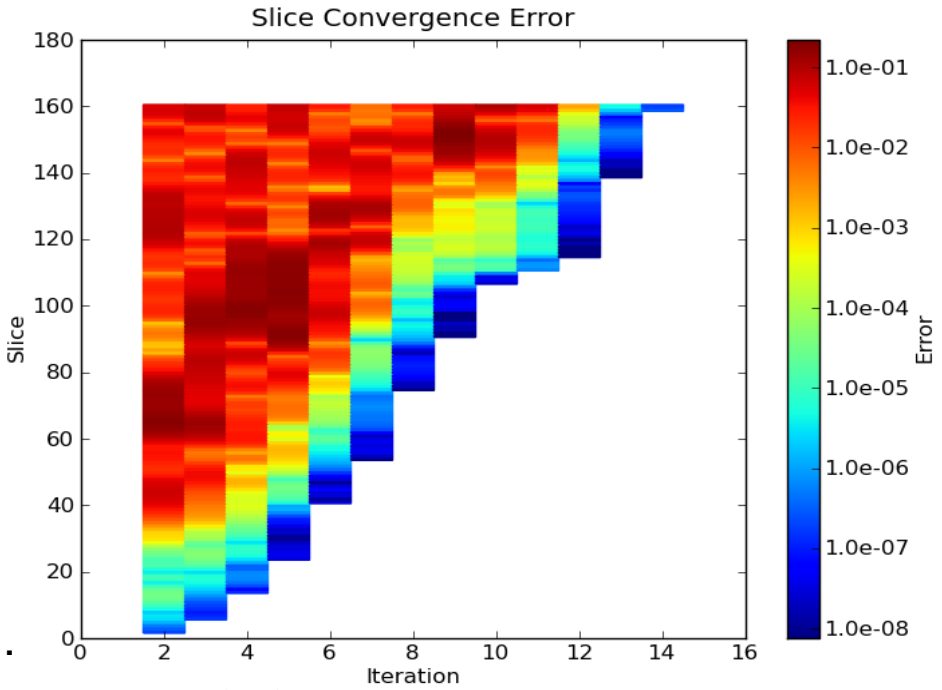
- Component level **`checkpoint()`** / **`restart()`** methods.
- Driver initiates checkpoint operation by call to checkpoint_components().
  - `checkpoint_components(comp_id_list, time_stamp, Force = False, Protect = False)`
- Checkpoints are a collection of component-specific files defined in the configuration file.
- **`save_restart_files()`** and **`get_restart_files()`** services to archive and retrieve component checkpoints.

Framework Services
Event Management

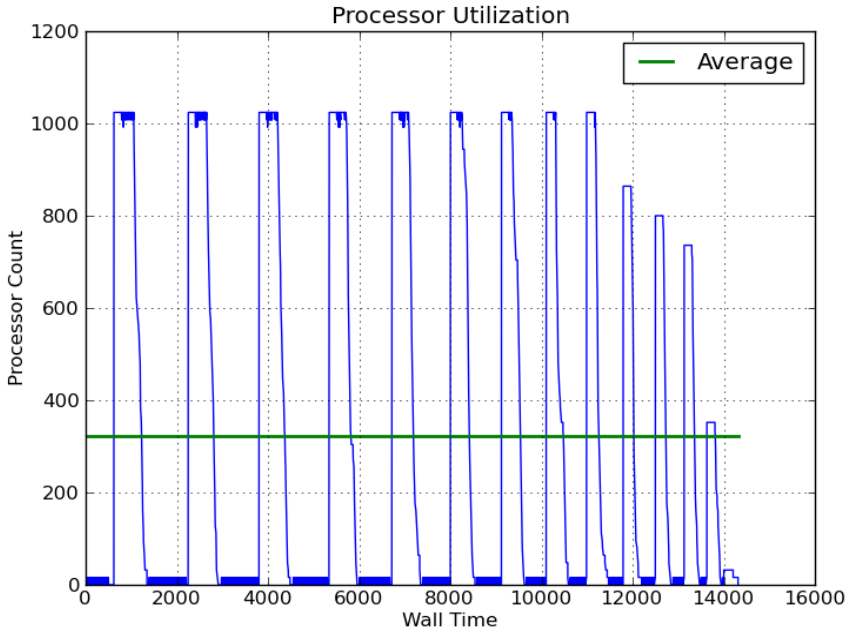
- publish(topicName, eventName, eventBody)
  -
```
event_data = {'key1': 'val1', 'key2': 345}
services.publish('Topic1', 'New Event', event_data)
```
- subscribe(topicName, callback)
  -
```
services.subscribe('Topic1', self.process_topic1)
```
  -
```
def process_topic1(self, topicName, theEvent):
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

Case Study: Parareal

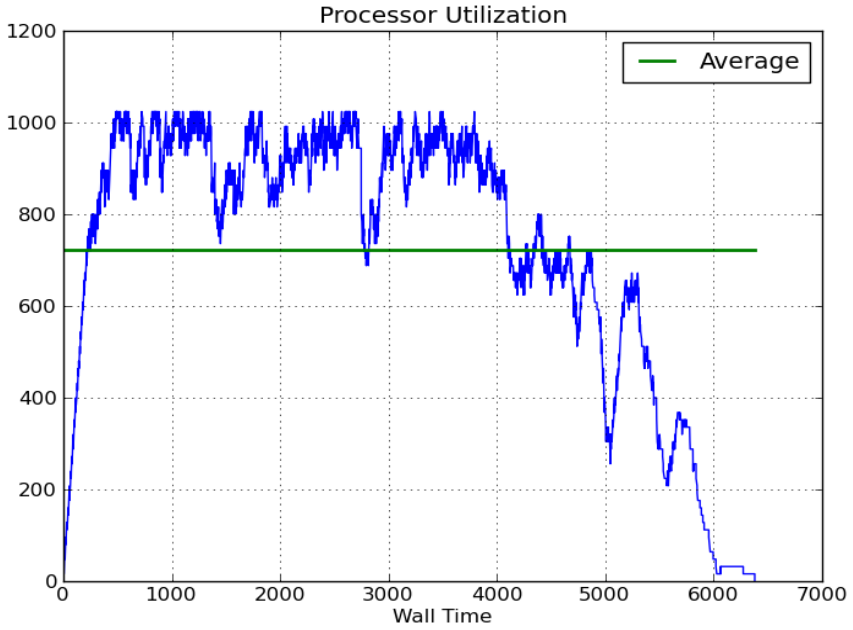
- Components:
  - Driver,
  - Coarse solver
  - Fine solver
  - *Converge test*
- Coarse & fine parallel tasks.
- Two implementations
  - "Deterministic" using task pool.
  - "Event-based" using asynchronous events to interleave coarse and fine tasks from different slices and itertaions.
- Template re-usable for other applications (with minimal changes).



Parareal Implementations



Using Task Pools



Using Asynchronous 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

- "***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

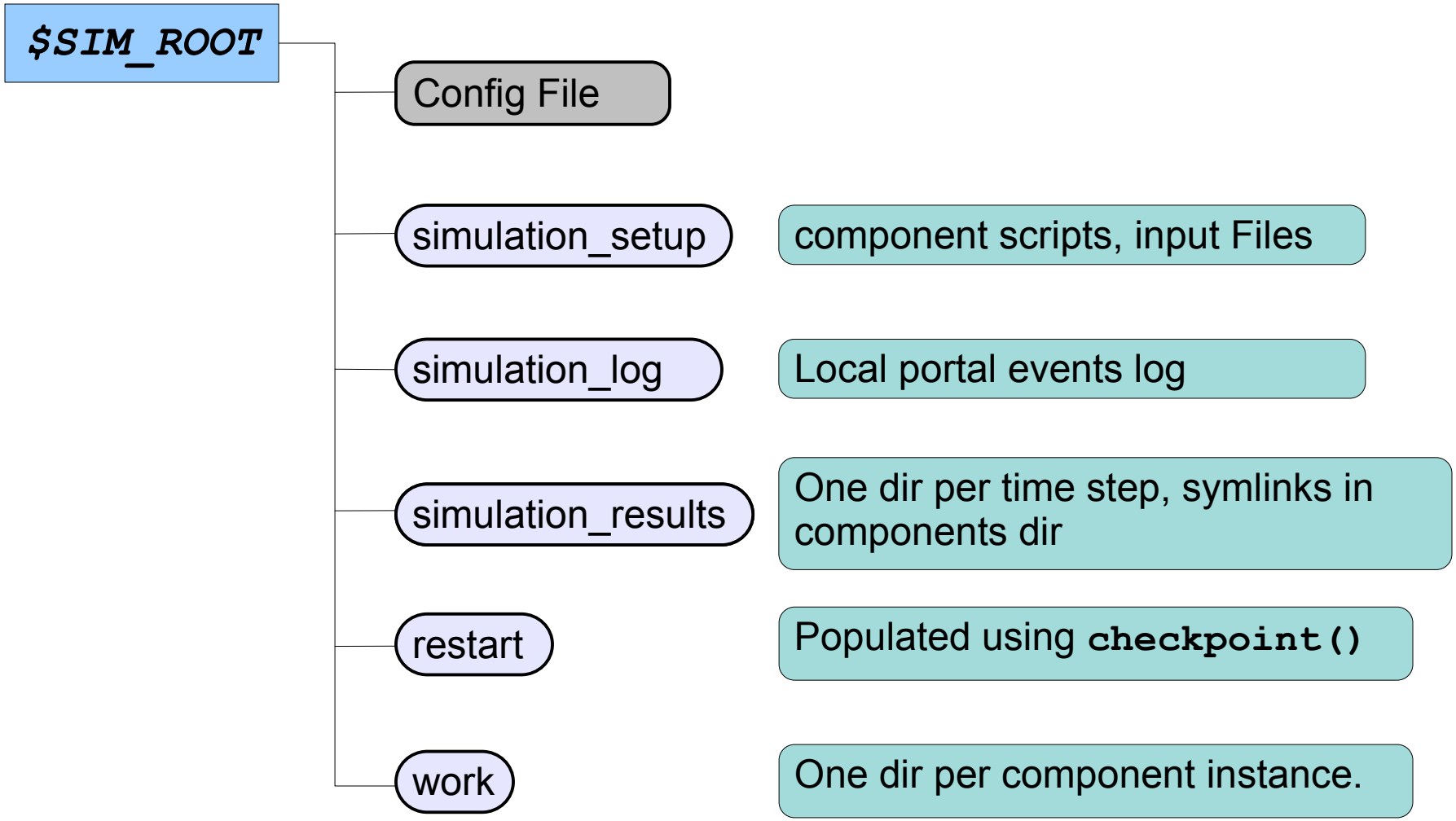
- Add `Makefile` and `Makefile.include` to the component directory
  - Copy from other component directories, and adjust path to **`IPS_ROOT`**, target binaries, scripts, and dependencies.
  - Compiler and default linked libraries (plasma state) specified in top level `Makefile.config` – included in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

Data Management
Simulation Tree Layout



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

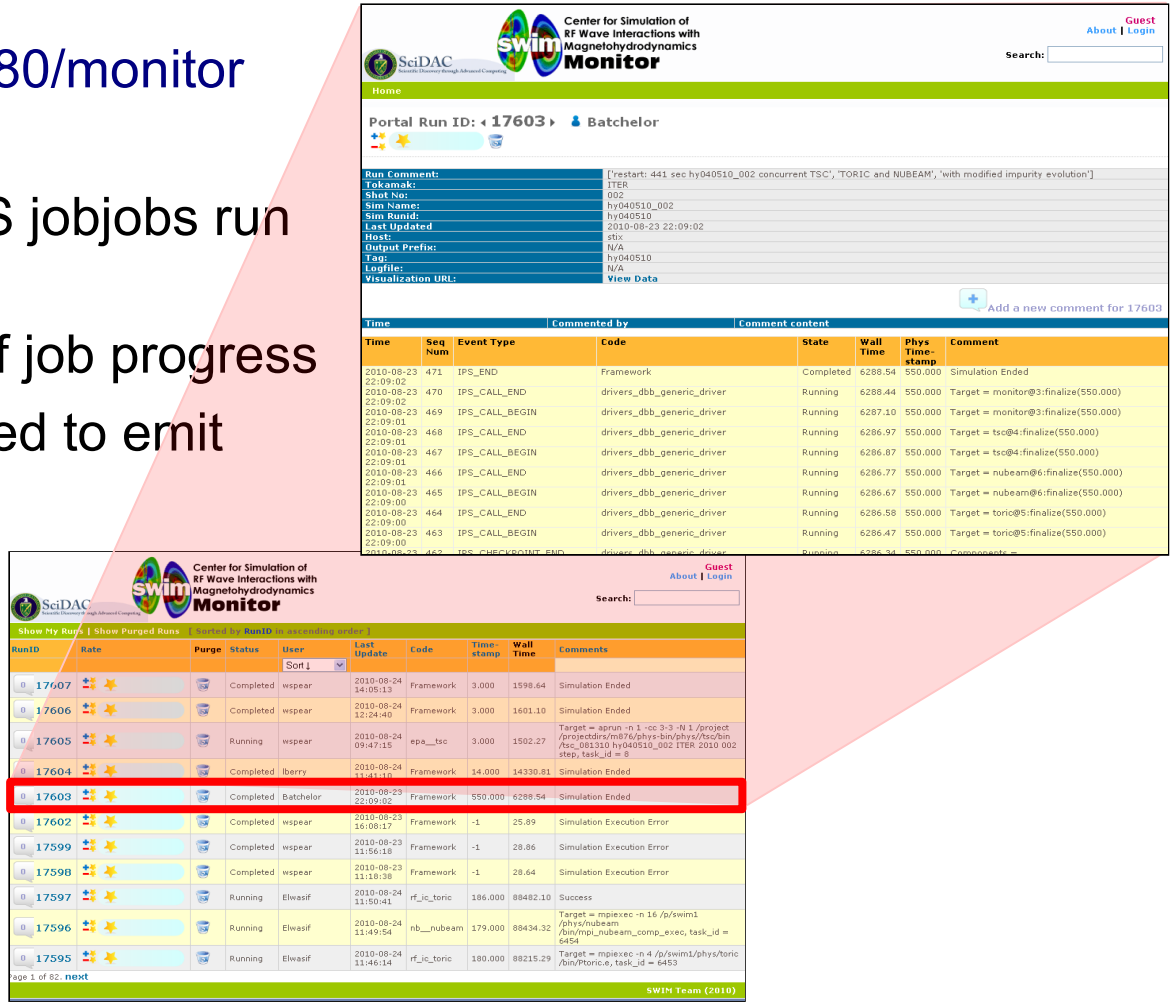
- Framework invoked in batch script, running on the *head node.*
- Multiple log files from batch submission:
  - Batch job stdout, stderr (one or two files)
  - Framework log file (from **`–log=`** command line option)
  - Simulation log file (one per simulation).

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.
- Monitor file accessed through the SWIM Web portal

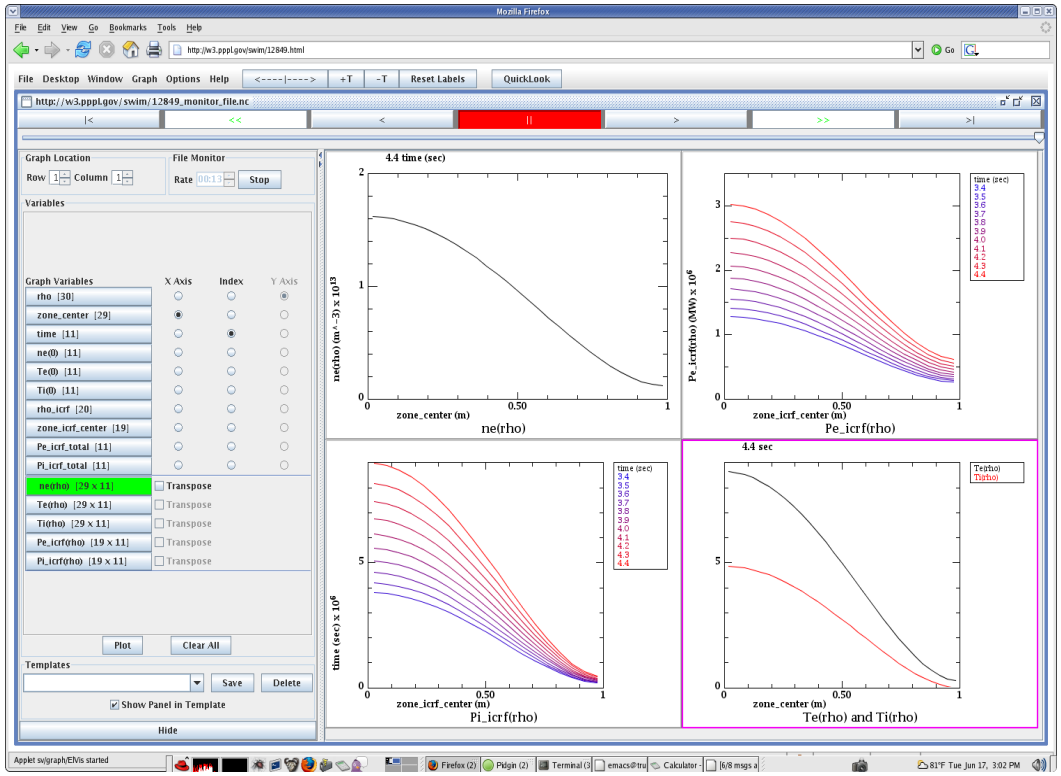
The SWIM Portal

- http://swim.gat.com:8080/monitor
  - Portal hosted by GA
- Portal use optional, IPS jobjobs run without it.
- Real-time monitoring of job progress
- Framework instrumented to emit progress events
  - Method invocations
  - Task life time
  - Data operations
- Components can send custom events.



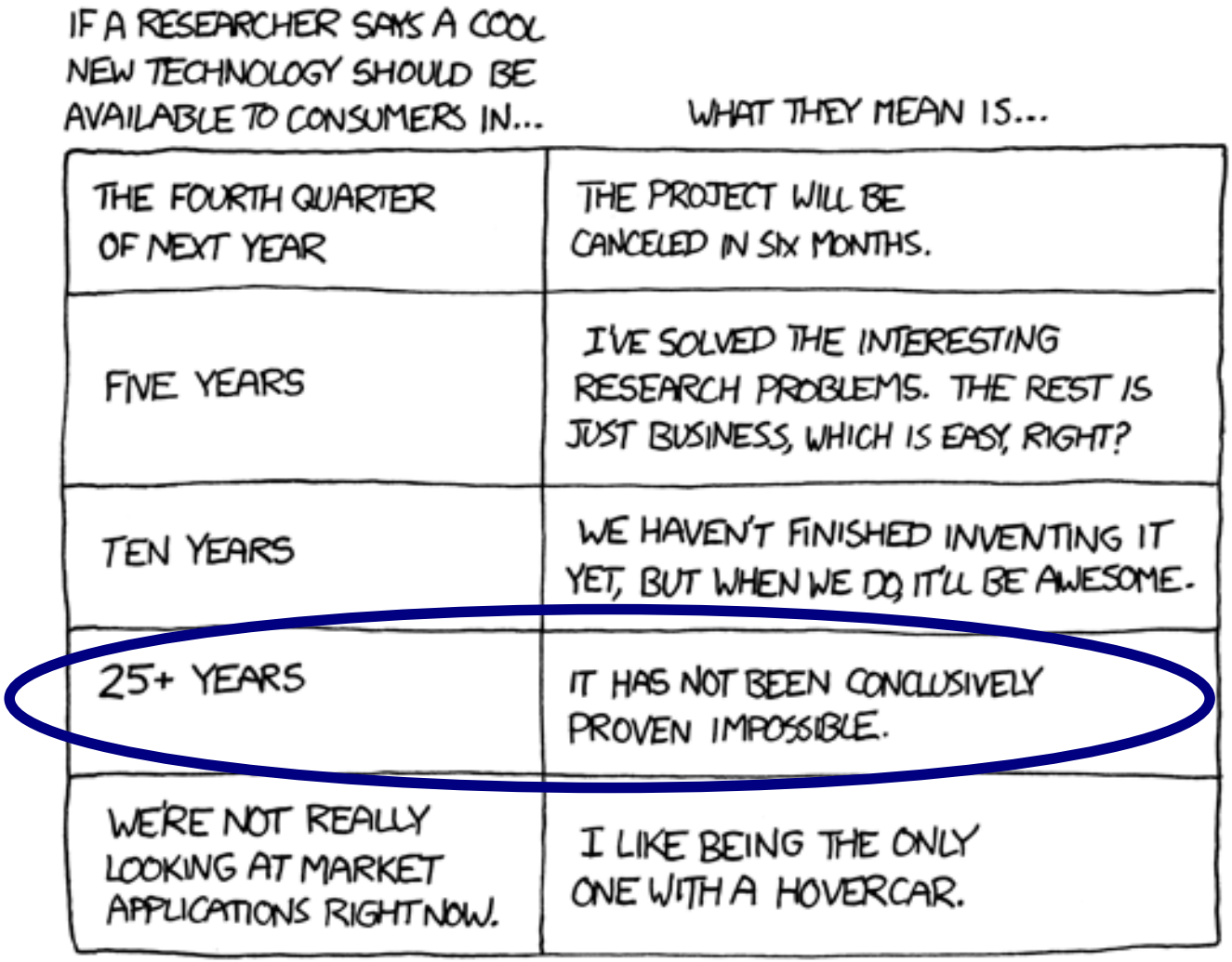
Real-Time Monitoring

- ElVis for quick monitoring
- Data generated using a monitoring component that is part of the running simulation.
  - Web accessible NetCDF file
  - Simulation history summary.
- Browser-based ElVis for short runs.
- Standalone version for long ones.



Ongoing & Future (Framework) Work

- Component input validation.
- More robust data management, and analysis.
- Improve documentation and ease of deployment.
- Component-as-a-server to reduce I/O overhead (when needed).
- Explore in-memory coupling options
  - Possibly using ADIOS


IF A RESEARCHER SAYS A COOL NEW TECHNOLOGY SHOULD BE AVAILABLE TO CONSUMERS IN...
WHAT THEY MEAN IS...
THE FOURTH QUARTER OF NEXT YEAR
THE PROJECT WILL BE CANCELED IN SIX MONTHS.
FIVE YEARS
I'VE SOLVED THE INTERESTING RESEARCH PROBLEMS. THE REST IS JUST BUSINESS, WHICH IS EASY, RIGHT?
TEN YEARS
WE HAVEN'T FINISHED INVENTING IT YET, BUT WHEN WE DO, IT'LL BE AWESOME.
25+ YEARS
IT HAS NOT BEEN CONCLUSIVELY PROVEN IMPOSSIBLE.
WE'RE NOT REALLY LOOKING AT MARKET APPLICATIONS RIGHT NOW.
I LIKE BEING THE ONLY ONE WITH A HOVERCAR.