

US National Virtual Observatory

# Scalable Framework Prototypes

Garching, June 2006

Doug Tody (NRAO/NVO/IVOA)



# System Framework – Introduction

- **Context**
  - Architecture – agreed
  - Technology evaluations – largely completed
  - Prototyping, more detailed design – *where we are now*
- **Approach**
  - *Separate implementation from interfaces*
  - Prototype using selected candidate technologies
- **OpenRTE Prototype**
  - OpenMPI/RTE selected for framework prototype in technology evaluation
    - Dozens of candidate technologies were reviewed, several in detail
  - Review what OpenRTE provides
    - Most of the functionality described is needed no matter what we use
    - Will not say much about MPI, but this is also needed for scalability
    - ORTE is complex, but basic usage is very simple, e.g.: `mpirun -np 4 myapp`
  - Subsystem Structure and Interfaces
    - Examine key subsystems in more detail
    - How do they look if mapped onto OpenRTE?

# OpenRTE Framework

(separate presentation)



# Subsystem Structure and Interfaces

- **Framework Subsystems or Key Interfaces**
  - Python Integration
  - Package Manager
  - Parameter Mechanism
  - Container
  - Messaging
- **Examine**
  - What each subsystem/interface does
  - Expected implementation using OpenRTE

# Python Integration (or any SL)

- **Capabilities Required**

- Load/unload package (globally shared)
  - Autogenerate Python bindings for all tasks/components
- Access package and parameter metadata (globally shared)
- Access parameter data (globally shared)
- Synchronously or asynchronously invoke task/method
  - Support for data and algorithmically parallel tasks

- **Implementation**

- Package Manager
  - global service to load/unload packages, manage paramfiles
- Task/method invocation

# Python Integration

- **Package management**
  - Load package
    - call PM to load the package, then do a bind package
  - Bind package
    - create Python bindings for all tasks in the package
    - task and parameter data (PM) used to autogenerate bindings
- **Task binding**
  - Task invocation
    - task can be called as a Python function
    - paramfile used to do parameter resolution (defaults etc.)
  - Task utilities
    - help, param editing, etc. immediately available for loaded pkg

# Python Integration

- **Head Node Process (HNP)**
  - The Python session is the HNP
  - All GPR data cached in-memory in Python/HNP process
- **Task execution**
  - Create per-execution paramset instance
    - base paramset, edit in values of command line arguments
  - Execute task via execution framework
- **Parallel tasks**
  - Tasks may be serial (uniprocess) or parallel (multiple processes)
  - Parallel tasks execute as a logical process group via framework
  - Resources permitting, all processes execute concurrently
  - For a parallel task each task gets a slightly different paramset
    - e.g., each task gets a different dataset to operate upon
  - This is normally controlled at the Python scripting level
    - e.g., list the datasets to be processed concurrently



# Package Manager

- **Component Package**
  - Global package metadata (XML)
  - Parameter set definitions (XML)
  - In-line documentation (XML or HTML)
  - Packaging (JAR/ZIP or something similar)
- **Package Manager**
  - Manages dynamically loadable application packages
  - API for managing (load/unload) and querying packages
  - Global shared storage for package and parameter metadata
- **API Example: Load Package**
  - Read package metadata, parse and load data into GPR
  - Load and register all parameter files
  - Tasks can then be executed and paramfiles accessed
  - Package metadata may be replicated, cached using GPR mechanism

# Parameter Mechanism

- **Parameter sets**
  - Parameters are organized into parameter sets
  - A parameter set is defined by a custom schema
  - A parameter set instance is a set of keyword,value pairs
  - Parameters may be scalar or vector quantities of any basic type
- **Types of parameter sets**
  - Task input params, task output params, simple data entities
- **Implementation**
  - A parameter set instance is stored in a GPR container
  - An API is provided for managing and querying parameter sets
  - Parameter data is cached for efficient access in a Python session
  - Concurrent access is provided via replication
    - e.g., a change in a Java GUI is visible in a Python session

# Container

- **Capabilities Required**

- A lightweight mechanism for invoking interchangeable modules
- Support multiple command protocols, e.g., host, ORTE, SOAP
- Provide the ability to package multiple components together
- Define the runtime environment seen by a component
- Define the external interface used to invoke a component

- **Execution modes**

- "host" – arguments on process command line (e.g., unix shell)
- "connected" – container is connected to and driven from framework

# Container Implementation

- **"Host" execution mode**
  - Framework (e.g., ORTE) is not used
  - Parameters specified on command line or in a file
  - Arguments specified on command line override defaults in file
- **"Connected" execution mode (using ORTE)**
  - Special arguments tell how to connect to ORTE session
  - Task/component to be run is specified on command line
  - At startup container connects to execution framework (ORTE)
  - Sets up the runtime environment for the named task
  - Accesses the paramset for the named task
    - Parameters are passed in binary form in a GPR container
    - All parameter resolution is performed before the task is called
  - Messaging is performed via the connected framework

# Messaging

- **Capabilities required**

- Standard i/o streams
- High bandwidth point-to-point streams
- Asynchronous publish/subscribe messages or events
- Logging (a variation on publish/subscribe)
- MPI connection groups (for parallel applications)
- Synchronous or asynchronous requests (RPC)

- **Implementation**

- ORTE runtime messaging layer (RML) provides most of these
  - This is separate from MPI messaging subsystem
  - Publish/subscribe mechanism tied to GPR (needs further analysis)
- I/O capture, redirection, and buffering already provided
- Full MPI integration already provided

# Plug-In Components

- **Capabilities required**
  - Link container directly into application layer
    - e.g., on a separate thread in a Python session
  - Ability to dynamically load shared library type components
  - Ability to call component methods directly from Python
- **Implementation**
  - Python binding wants to call component method directly
  - Framework (ORTE) not involved in most method calls
  - Framework still used for messaging, parameter access
  - A task (parameter-based) method works normally except that
    - parameter data does not have to be transmitted
  - Low level (CORBA/IDL-like) methods are also possible
    - in this case Python C-API needs to build argument list
    - component interface defined in XML used to drive this
    - C method of component can then be called directly
    - direct access to arrays in memory is possible