Astronomical Data Analysis Software and Systems XX
ASP Conference Series, Vol. 442
Ian N. Evans, Alberto Accomazzi, Douglas J. Mink, and Arnold H. Rots, eds.

New Control System Software for the Hobby-Eberly Telescope

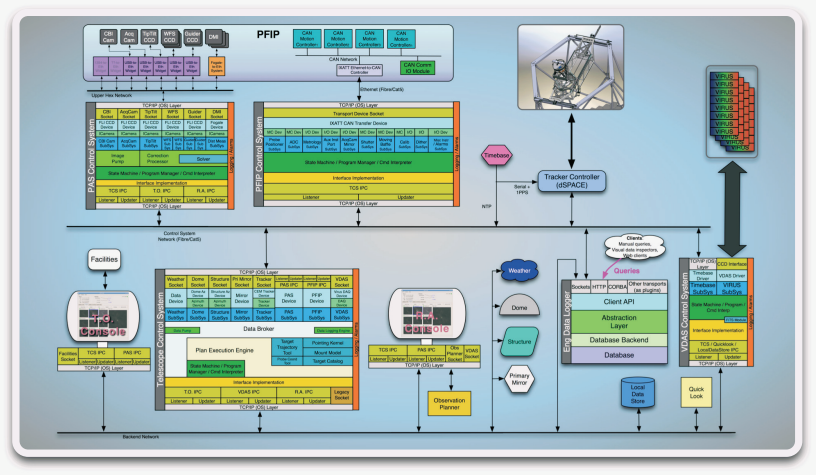
Tom Rafferty,* Mark E. Cornell, Charles Taylor III, and Walter Moreira

McDonald Observatory, University of Texas at Austin, 1 University Station C1402, Austin, TX, USA 78712-0259

rafferty@astro.as.utexas.edu, thomasrafferty@gmail.com

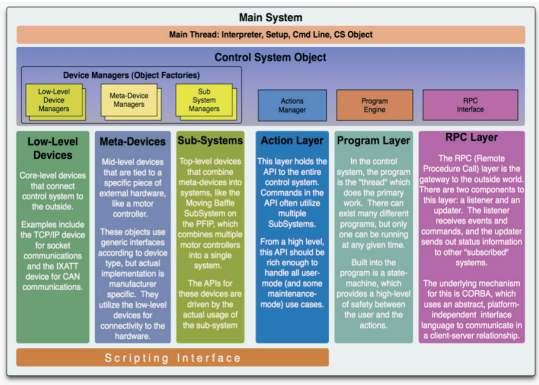
Abstract. The Hobby-Eberly Telescope at the McDonald Observatory is undergoing a major upgrade to support the Hobby-Eberly Telescope Dark Energy Experiment (HETDEX) and to facilitate large field systematic emission-line surveys of the universe. An integral part of this upgrade will be the development of a new software control system. Designed using modern object oriented programming techniques and tools, the new software system uses a component architecture that closely models the telescope hardware and instruments, and provides a high degree of configuration, automation and scalability. Here we cover the overall architecture of the new system, plus details some of the key design patterns and technologies used. This includes the utilization of an embedded Python scripting engine, the use of the factory method pattern and interfacing for easy run-time configuration, a flexible communication scheme, the design and use of a centralized logging system, and the distributed GUI architecture.

1. Overview



The new control system architecture consists of a closely coupled group of distributed systems. Each system is responsible for specific functions based on type or proximity to hardware, and is designed to be run autonomously. A simple but flexible messaging scheme allows communications between the systems.

2. Control System Architecture



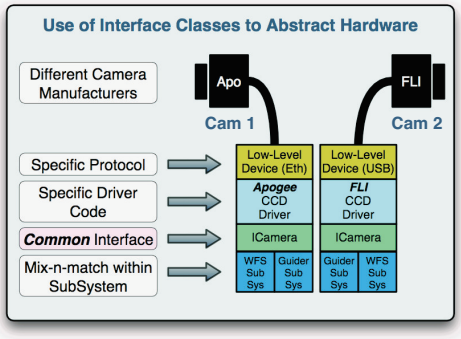
Phased Startup:

1. Main thread starts the server, reads command-line args, creates the context object and interpreter
2. Control System object creation, managers are created and setup with device via script
3. Pre-wiring: All managers instantiate their devices and device objects get pre-wiring setup
4. Wiring: Managers "wire" all their devices
   - All low-level devices establish communications with input/output hardware
   - Meta-devices are configured to talk to their respective low-level devices
   - Sub-Systems are configured with Meta-devices
   - Post-wiring configuration of devices
5. Actions layer enabled
6. CORBA/RPC layer enabled
7. Default program is loaded and starts to run

All major control systems use the same basic architecture, highlighted above. Each control system has a collection of managers, or object factories, whose job is to create and destroy all device objects in the system. They are configured at run-time by the embedded scripting engine. The API to the control system consists of a set of actions, which are controlled by a high-level, state-machine based program called actions processing. This allows asynchronous control of the underlying hardware.

3. Design Principles

Flexibility, along with reliability, were two of the primary drivers of the design. We are using modern, yet proven, programming techniques, utilizing a common tool chain and widely available libraries.



- Component oriented design
- Object oriented programming techniques
- Physical world is composed of components and modules that are connected using defined interfaces
- Common code throughout
  - Provides base classes for all objects in system
  - Similar hardware drivers implement the same generic interface class
  - Shared algorithms, routines, utilities, etc
- Scriptable interface built in at device level
- Each individual control system:
  - Controls low-level device drivers to provide I/O to the hardware
  - Provides abstraction interface to the devices so changing hardware manufacturer is simple
  - Defines it's API using Actions with a marshaled interface to the action processing system
  - Run programs that use actions to manipulate the device sub-systems
  - State machine driven

4. Scripting

At the heart of the control system lies an embedded Python scripting interface. All devices, subsystems, and actions are available to scripting engine through the "magic" of SWIG wrappers. The control system is configured entirely from within scripting layer which executes shortly after startup. This allows the capability to completely change the configuration of your system simply by using a different or modified configuration script. Testing/debugging is simplified because you can easily isolate any driver or change its properties. Program code can start out as

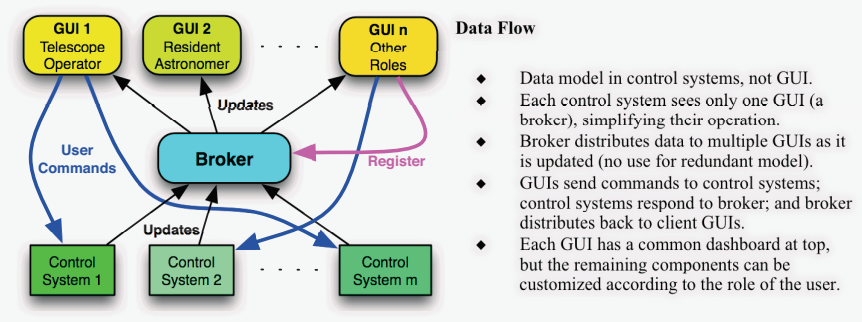
a prototyping script to allow agile development without changing any of the core driver code or algorithms. The use of Python as interpreter brings in a very rich and well tested set of modules for extending the base functionality of the control system, with little or no modification to the base code.

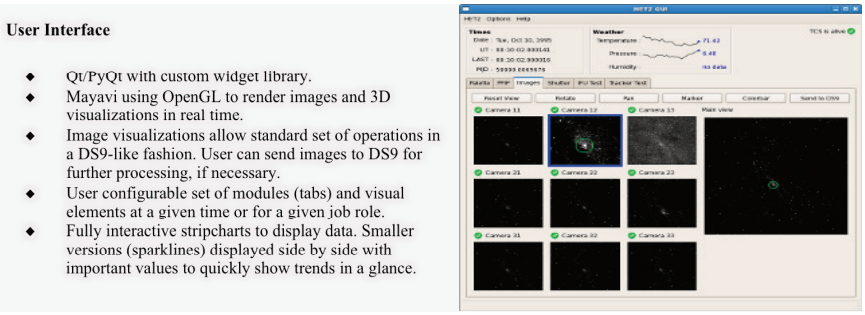
5. Communications and Messaging

In order to keep the communications and messaging between components and major systems as flexible as possible, we use JSON documents. JSON (JavaScript Object Notation) is a lightweight, data-interchange format that is easy for humans to read and write. By using JSON as the payload in our communications, we can easily add or modify commands without having to re-factor the interface.

6. GUI

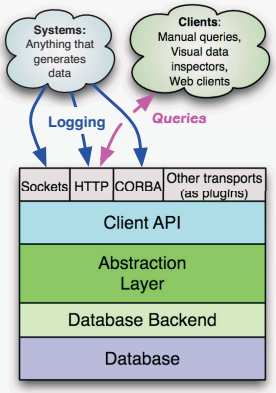
The design of the GUI system allows multiple individual GUI screens to exist at any given time. Each can be configured differently based on the role of the user. The data model for the GUI resides in the control systems; therefore there is no disconnect between different GUIs that are running simultaneously.





7. Logger

The logging system is intended to be an obervatory-wide system available 24/7 to a wide variety of data-producing systems using a variety or protocols. Command-line as well as web-based clients can be used to extract data from the system.



Network Server:

- 24/7 availability allows high number of concurrent connections.
- All major system connect to it via sockets, http, etc.
- Uniform API for all the protocols.
- Separation of network server and database engine processes through a fast internal synchronized queue.
- Uses JSON format for logging and querying, and for returning set of results.
- Supports raw unformatted strings for clients with limited computing power.

Database Backend:

1. Uses schema-less design (NOSQL).
2. Database engine can be changed by writing a thin layer to adapt to a defined interface.
3. Currently uses MongoDB allowing high insertion speed and easy distributed replication.
4. Separation of logging and querying OS processes to minimize interference in case of overloading. Each of them can run on separate machines if necessary.

Abstraction Layer:

1. Handles the logic between network and dB backend.
2. Does stemming analysis on text to allow fuzzy queries (plurals, conjugated verbs, etc.).
3. Adds data about the client and timestamps if necessary.

8. Development Environment

The development team uses the usual list of open source tools and libraries as described in the following table.

Primary Platform	RHEL 64-bit
Languages Used	C/C++, Python, tcl/tk
Toolchain	gcc, gnu make, bash, lapack, libtool, automake
Technologies	SWIG, omniORB, omniORBpy, SLALIB, pySlalib, CFITSIO, pyFits, libedit, sqlite, cJSON, log4cplus, Qt4/pyQt, Mayavi/OpenGL, Chaco, NumPy, SciPy, VTK, SetupDocs, Pyrex, Sphynx, Greenlet, GSL, MongoDB, Stemming
Continuous Integration	Nightly builds using Hudson
Version control	git with a central 'master' repository
Other	Automatic building of auxiliary libs; use of bugzilla for bug tracking; wiki used to capture specs, brainstorm amongst developers, document processes

9. Further Information

To obtain the original ADASS XX (2010) poster in pdf form, please use the following URL: `http://www.philomather.com/ADASS_XX_P060.pdf`. The author can be contacted via email at `rafferty@astro.as.utexas.edu` or `thomasrafferty@gmail.com`.