

Configurable Monitoring in the Flux Resource Manager

Student: Samuel Heinz, UC Santa Barbara
Mentor: Jim Garlick, Lawrence Livermore Nation Laboratory

Special Thanks To: Becky Springmeyer, Dong Ahn, Mark Grondona, Don Lipari, Stephen Herbein, and Tapasya Patki

What is Flux?

- Flux is a next generation Resource Management (RM) software designed for High Performance Computing (HPC).
- Current RM software and approaches will likely not be able to overcome the ever-growing challenges as we approach Exascale computing. Flux intends to address these challenges by treating entire HPC centers as a single common group of different resources.
- Flux will be a scalable framework, allowing for site-wide constraints (e.g. limits to power, IO, and, cores), while still allowing for effective scheduling strategies.

Modules

- Flux's architecture is one based on a set of unique services named Comms Modules.
- Flux core is only a communication broker between adjacent nodes and its loaded modules.
- Flux has numerous modules, each performing unique services for the framework.
- This poster discusses the functionality of the module responsible for monitoring.

Monitoring

- The Monitoring Module is itself a framework.
- It allows user's of Flux to load multiple, unique plugins.
- These plugins can be chosen from a set of defaults's.
- It also allows for new plugins to be written, compiled, and easily loaded.

Plugins


- Plugins are shared object files that declare three functions.

Plugins are triggered by a broadcast sent to its local monitoring module. The module then executes the first function for all loaded plugins.

Source() This function gathers its specified data. It can optionally assemble the data into a specific format. It then returns this aggregated data to its local module as a **packet**.

Data:  **Aggregated Data:** 

Aggregated Data for each plugin is called a **Packet**: 

Aggregated **Packets** from separate plugin's are called **Bundles**: 

The module then begins listening for **bundles** from all its child nodes. It waits until all have been gathered (or, optionally, the wait has timed out). The module then opens these **bundles**, and distributes the **packets**. Each plugin is passed the locally sourced **packet** as well as any recieved **packets**. It is then that the next function is called for each plugin.

Reduce() This function can vary wildly between plugins. It could simply append the local **packet** to the other's. Alternatively it could implement an intellegent reduction algorithm. Th algorithm could compress and keep only the desired data. This allows **packet** sizes to remain small even in extremely large systems.

The module waits until all plugins have returned their newly compressed **packets**. The module then masses all **packets** together into a single **bundle**. Most instance's of the module now simply pass the bundle up to its parent node. It is only for the master node (e.g. Node 0) that the final function is called.

Sink() This function is how the data is actually written out. The master node is now in possession of a single **bundle**. This **bundle** contains every plugin's packet, reduced from every node. The user is in control of how this data is written out. The **bundle** could be written to stdout, saved as a tmpfile, or even discarded.

