Automated Job Monitoring in a High Performance Computing Environment

Robert F. Cromp
Gilad Suberri
Raytheon Company

robert_f_cromp@raytheon.com
gsuberri@eos.east.hitec.com

Abstract

We are developing software that monitors High Performance Computing assets while users’ batch jobs execute, and actively performs site-established corrective actions to handle routine system/queuing issues normally performed by Unix administrators. The Automated Job Monitor is independent of both platform and queuing system, and is customizable for numerous domains.

1. Introduction

A typical high performance computing (HPC) environment is quite complex, providing hundreds of active users access to hundreds or thousands of processors. Computational efficiency is best achieved by running jobs in batch mode—dedicating resources to a specific job for an established time, thus minimizing the time spent swapping processes and large volumes of data in memory, and eliminating problems with coordinating multiple processors from one job’s time share step to its next.

To assist in scheduling these jobs, system administrators provide specific queues with fixed properties. The queuing system matches each job to the appropriate queues based on user-supplied constraints.

The system administrators need to ensure that each HPC platform sustains a high work load, and that each submitted job eventually has its resources allocated so that it executes. High performance computing centers typically have a large number of jobs pending, and a job is usually not reviewed by a system administrator until either a user questions why it has not started, or a system administrator notices that it has been pending an unusually long time. Pinpointing the reason can be time-consuming, due to the complexity of the compute environment and the multitude of scheduling policies in play.

2. System Design

The Environment contains all the real time data that the application will monitor. This includes, for example, all queued jobs, running processes, and file system information. To analyze this data, a snapshot is taken of the Environment and the results are stored in Working Memory. To assure the data is current, Working Memory is updated with a fresh snapshot of the Environment at a configurable frequency.

The Inference Engine monitors the Working Memory for changes. When a piece of information is updated with a new value, the Inference Engine collects all rules from the Rule Set that apply to the updated data. These rules contain an antecedent and a consequent and thus can be represented as “if x is true, then do y.” The set of applicable rules is called the Conflict Set and is analyzed by the Conflict Resolver, which selects which rules to fire and assures that no conflicting rules are evaluated. Information about the rules fired is logged to a History Database to assist in gaining a better understanding of the system.
2.1 Readers

A Reader is the entity that transfers information from the Environment into Working Memory. Each Reader knows what system calls need to be executed to acquire the data it is responsible for transferring, how to parse the information, and how frequently to update Working Memory. The Reader parses the data into one or more Working Memory Objects. These objects are simply a list of attribute-name/value pairs that define the Environment’s data.

A scheduler executes the Readers at their specified frequencies. Two schemes have been developed for this. The first scheme executes each Reader at its configured frequency, allowing the Reader to complete its system call, parse the data returned, and update Working Memory before the next Reader is executed. Though in most cases this scheme is sufficient, a possible issue arises if two Readers are scheduled at the same frequency. If the first Reader to execute takes x amount of time to complete execution, the next Reader will be x amount of time late in its execution. This could cause unfavorable results.

The Producer-Consumer scheme addresses this issue. In this method, the Reader’s job is split into two sections. The Producer is responsible for the system call and storing the data in a buffer. The Consumer takes the data stored by the Producer, parses it, and inserts it into Working Memory. The Consumer is constantly running and checking for new data in the buffer; the Producer runs in respect to the frequency of the Reader, and only executes the system call before returning control to the scheduler. A Producer’s system call does not even need to complete running before control is returned to the scheduler, giving the application more accuracy in terms of executing Readers at their respective frequencies.

A Reader could also be called on an as-needed basis. For instance, another part of the system may realize that a certain piece of information critical in making a decision is missing from Working Memory. The AJM will attempt to find and execute a Reader to get this specific data.

2.2 Rules

Rules are composed of an antecedent and a consequent. An example would be, “If a running job is using more processors than it requested, then kill the job and notify the system administrator of the job’s details via email.” When an attribute’s value changes, applicable rules are evaluated, thus allowing the AJM to perform site-established corrective actions. A rule is considered applicable to an updated attribute of Working Memory if its Read Set contains the attribute being updated. The Read Set of a rule is defined as the set of attributes that are being evaluated, or read from, in the antecedent. The Conflict Set is a set of rules such that the Read Sets of all of these rules contain the attribute updated.

A Write Set is also associated with each rule. The Write Set of a rule is defined as the set of attributes the rule’s consequent will write to and therefore possibly change. The Conflict Resolver can use the Write Sets of the rules in the Conflict Set to decide which ones to evaluate.

An issue concerning the Write Set of a rule is as follows. Assume Rule1 and Rule2 are in the Conflict Set, with the Read Set(Rule1) = {a}, Read Set (Rule2) = {a, b}, and the Write Set(Rule1) = {b}. If Rule 1 is evaluated first and changes the value of b, this could change the outcome of whether or not Rule 2’s antecedent is evaluated to true, though it is part of the Conflict Set corresponding to attribute a, not attribute b. Therefore, any changes rules make explicitly to attribute values are “local” changes, until all the applicable rules have been evaluated and then the “global” Working Memory can be updated.

3. Configuring the System

We have developed a graphical user interface using wxPython to assist the user in configuring their specific monitoring scenario. The Class Builder is used to define the structure of all the Working Memory Objects. The Rule Builder configures the system’s rules. The user defines the antecedent of the rule by referring to objects defined in the Class Builder along with common logical and mathematical operators. Finally, the Reader Builder creates and configures the system’s readers. The Builder analyzes the output for different ways to parse, including common delimiters and fixed width, and displays the different possibilities. After a parsing style is selected, the user sets the different pieces of the parsed output with their corresponding attribute names. After this process is completed (one time only per command), an XML representation of the reader is created.

References