Designing and Implementing a Classad Viewer for the Condor Distributed System

Mamadou B. Diallo

Abstract

Classads are an important component of the Condor Distributed System. They carry a wealth of information about machine attributes and their values. One disadvantage of classads is that they do not have a graphical user interface to navigate them. It has now become a necessity to develop such an interface for viewing classads' content. The present project designed and implemented the ClassadViewer. The interface was designed following a typical window based browser, with tree structures, tables, menus and menu bars and is primarily accessed via mouse events. The interface is highly customizable to a user's preferences. This customization includes what to view and how to view it. The tool is compact, portable, and easy to use. It also performs and responds, from the user's stand, as fast as a commercial operating system file browser.

1 Introduction and Background

The Condor distributed system is perhaps one of the earliest peer-to-peer computing systems in the US. It exploits the processing power of idle workstations and makes it available to applications that require tremendous amounts of processing. Such applications include areas in Simulation, Molecular Biology, Physics, Chemistry, and Astronomy to name a few. Users in these fields can submit their computationally intensive jobs to Condor via Requirements Expressions. These expressions simply allow users to specify for each attribute, what value or range of values to look for. For instance, a user may specify that his computation must run on workstations having at least 64 MB of memory via the expression "Memory >= 64". The set of workstations that is available to Condor is known as the Condor Pool. Each of these workstations has attributes (such as Architecture, Memory, Operating System, and so on) that must be advertised. Classified Advertisements (classads for short) are used to describe these workstation attributes. In this case, the classads are referred to as machine classads. Classads can also be used to describe jobs (job classads) and to inquire about the current state of the system. Upon submission of a job to the Condor system, the user's requirements are compared with the actual attribute values of the machines in the Condor pool, and a candidate workstation (or set of workstations) is selected to run the job.

Traditionally, users have been limited to a command-line interface and text based output for viewing classads. However, as the number of workstations in the Condor pool grows, and as a consequence, as the number of classads grows, a need has arisen for an application that can be used to navigate through the maze of classads and lets users classify them in views that are meaningful to them. This task will facilitate identifying for instance whether some machine exists that meets a user's requirements and what to fall back on if it does not. The aim of this project is to design and implement such an application.

The application will present to the user all the classads that exist in the Condor pool. These will be displayed initially in some default configuration. The application will then allow the users to customize this default configuration, by adding new views or deleting undesired ones, choosing how to display the data, and saving the customized view for the next session.

2 Design Idea

The application's design idea revolves around using a tree structure to hold and manipulate the classads. This idea was proposed early in the design by Jaideep . Each node of the tree represents an attribute. Attributes can be viewed as child nodes of a root node, which holds all the classads collected. By default, this root node has two child nodes: 'OpSys' and 'Arch', each of which groups all the classads into Operating System and Architecture respectively. Besides these two, users can add any other attributes to the root node, or to the 'OpSys' and 'Arch' nodes. Adding an attribute functionally means grouping by that attribute all the classads defined for that attribute's immediate parent. For instance, the 'SOLARIS28' value under the attribute 'OpSys' can further be grouped by Memory, thus dividing all the SOLARIS28 classads into 64MB, 96MB, and so on. Each value under Memory can then further be grouped by another attribute (such as LoadAvg). This process can be repeated until no new groupings are possible. Each of these added nodes will display fewer and fewer values. In other words, these nodes allow for classifying the classads from the more general point of view to the more detailed and specific point of view. For example, a user can create a view to display all SOLRARIS28 machines with 256 MB of Memory, that are Idle and whose LoadAvg is 0. Deleting attributes from the tree produces the reverse effect.

In a similar manner, the data displayed in the table is displayed in a default manner. Each row in the table represents a classad, and each column in the table represents an attribute. By default, only the following eight attributes are shown in the table: Name, LoadAvg, Memory, Arch, OpSys, KFlops, State, and Activity. The user can then add any other attributes by adding new columns to the table. Columns can also be removed from the table. Values in the table can be configured by users and made more specific up to a desired format. For example, real numbers are displayed with a default of three decimal places, allowing to be formatted anywhere from no decimal place to scientific notation. The next section gives the procedural view of the tree.

3 System Functional Specification

There are two major parts to this project. The first part addresses the question what attributes should be displayed and the second part addresses how to display the values related to these attributes. Perhaps the most important feature of this application would be how much it lends itself to customization. This is so since each user may have his or her own different way of viewing the different attributes in the Condor pool. For example, one user may not care to view the architecture attribute displayed, while this may be important to another. Each one of these users should be able to customize the application in such a way as to display only the attributes that are important to him or her. In a similar way, one user may wish to display all machine times in seconds, while another may wish to display them in hours, minutes, and seconds.

Before we move on to present the features of the application, it is appropriate to introduce a couple of important terms at this point.

An attribute: is a property that a machine possesses, such as architecture, operating system, memory, and state. At the writing of this report, the classads in the Condor pool of the CS Department at UW-Madison totaled 70 attributes. It is important to note that not all machines will have values for all attributes. In other words, it is okay if some attributes are missing. It is also important to note that new attributes can be added without warning at any time.

An attribute value: is the actual value that a given attribute has. For instance, the attribute 'OpSys' can take on values 'NT', 'SOLARIS28', 'Linux', and so on. If an attribute is not present on a certain machine, its corresponding value is set to null or undefined. These values are classified into six different types: UNDEF, ERROR, boolean, string, integer, and real. If an attribute does not exist on a machine or its value is not defined or its value is null, then the value will have type UNDEF. These values will be shown in the table as 'UNDEF'.

3.1 Features/Functions Performed

The project's requirements specification came along with the implementation. An iterative approach was taken in designing and implementing the features of the application. There was constant communication between Professor Miron Livny, Professor Marvin Solomon and me in identifying what features were needed and should be provided by the application. At the end of this iterative process, the application had reached a decent level of functionality and had the following features.

Attributes Displayed

There are two viewing areas provided by the application: a hierarchical view in the form of a tree (where attributes and their values are arranged), and a table view in the form of a spreadsheet (where the data corresponding to values are displayed).

Initially, all existing classads are collected and displayed in the tree (the default root tree called 'Machine classads') under two attribute nodes: 'Arch' and 'OpSys', with the number of collected classads prefixed to each. This is the default configuration of the application.

Any attribute can then be added as a child node of the default root tree or any of its subtrees. Each attribute value would in turn be a child node of the subtree rooted at the parent attribute. This hierarchy reduces at each level of nesting the number of classads to be displayed. Here again, the number of classads in which the given attribute is defined is prefixed to the attribute name. If there are classads in which an attribute is not defined, they will be grouped into a pseudo attribute value created on the fly and named <UNDEF> prefixed by their number.

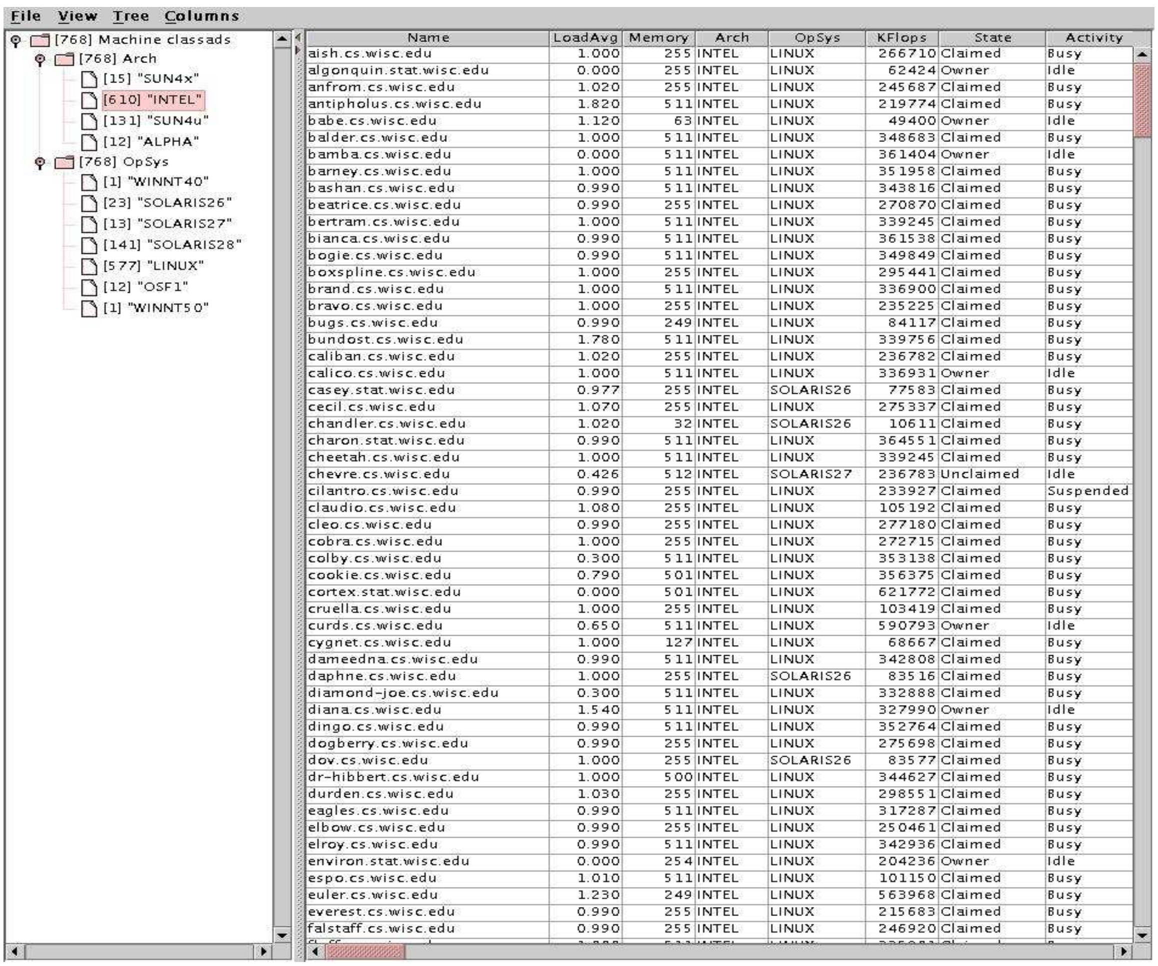
The user is allowed to add new nodes at any subtree, thereby creating new or custom views. If the chosen node is a subtree, then the attribute value is added to all the children of the subtree. Likewise, the user is able to delete any nodes from the default subtree that are not of interest to him or her.

On the table side, the user can add or delete columns from the table. Each table column corresponds to an attribute and the values in columns correspond to the given attribute's values from the classads.

After a long period of activity (or inactivity for that matter) with the current classads, the user can choose to refresh the data and get the latest classads from the collector. This is an important feature since attribute values change frequently in the Condor pool.

As with many graphical user interfaces, one of the nicest features of the application is its ability to be customized. The user can add or delete attributes from the views at will. Once the desired configuration is achieved, it can be saved as a preference for all subsequent sessions until the user chooses again a new configuration.

As long as a leaf node (i.e., an attribute value node) has not been selected in the tree, all the attribute value nodes' data are displayed. Once a leaf node (i.e., an attribute value node) has been selected, only values for that node are displayed. Following below is a screenshot of the table's default output, showing the initial eight attributes and displaying all classads for the Intel architecture.



Name	LoadAvg	Memory	Arch	OpSys	KFlops	State	Activity
aish.cs.wisc.edu	1.000	255	INTEL	LINUX	266710	Claimed	Busy
algonquin.stat.wisc.edu	0.000	255	INTEL	LINUX	62424	Owner	Idle
anfrom.cs.wisc.edu	1.020	255	INTEL	LINUX	245687	Claimed	Busy
antipholus.cs.wisc.edu	1.820	511	INTEL	LINUX	219774	Claimed	Busy
babe.cs.wisc.edu	1.120	63	INTEL	LINUX	49400	Owner	Idle
balder.cs.wisc.edu	1.000	511	INTEL	LINUX	348683	Claimed	Busy
bamba.cs.wisc.edu	0.000	511	INTEL	LINUX	361404	Owner	Idle
barney.cs.wisc.edu	1.000	511	INTEL	LINUX	351958	Claimed	Busy
bashan.cs.wisc.edu	0.990	511	INTEL	LINUX	343816	Claimed	Busy
beatrice.cs.wisc.edu	0.990	255	INTEL	LINUX	270870	Claimed	Busy
bertram.cs.wisc.edu	1.000	511	INTEL	LINUX	339245	Claimed	Busy
bianca.cs.wisc.edu	0.990	511	INTEL	LINUX	361538	Claimed	Busy
bogie.cs.wisc.edu	0.990	511	INTEL	LINUX	349849	Claimed	Busy
boxspline.cs.wisc.edu	1.000	255	INTEL	LINUX	295441	Claimed	Busy
brand.cs.wisc.edu	1.000	511	INTEL	LINUX	336900	Claimed	Busy
bravo.cs.wisc.edu	1.000	255	INTEL	LINUX	235225	Claimed	Busy
bugs.cs.wisc.edu	0.990	249	INTEL	LINUX	84117	Claimed	Busy
bundost.cs.wisc.edu	1.780	511	INTEL	LINUX	339756	Claimed	Busy
caliban.cs.wisc.edu	1.020	255	INTEL	LINUX	236782	Claimed	Busy
calico.cs.wisc.edu	1.000	511	INTEL	LINUX	336931	Owner	Idle
casey.stat.wisc.edu	0.977	255	INTEL	SOLARIS26	77583	Claimed	Busy
cecil.cs.wisc.edu	1.070	255	INTEL	LINUX	275337	Claimed	Busy
chandler.cs.wisc.edu	1.020	32	INTEL	SOLARIS26	10611	Claimed	Busy
charon.stat.wisc.edu	0.990	511	INTEL	LINUX	364551	Claimed	Busy
cheetah.cs.wisc.edu	1.000	511	INTEL	LINUX	339245	Claimed	Busy
chevre.cs.wisc.edu	0.426	512	INTEL	SOLARIS27	236783	Unclaimed	Idle
cilantro.cs.wisc.edu	0.990	255	INTEL	LINUX	233927	Claimed	Suspended
claudio.cs.wisc.edu	1.080	255	INTEL	LINUX	105192	Claimed	Busy
cleo.cs.wisc.edu	0.990	255	INTEL	LINUX	277180	Claimed	Busy
cobra.cs.wisc.edu	1.000	255	INTEL	LINUX	272715	Claimed	Busy
colby.cs.wisc.edu	0.300	511	INTEL	LINUX	353138	Claimed	Busy
cookie.cs.wisc.edu	0.790	501	INTEL	LINUX	356375	Claimed	Busy
cortex.stat.wisc.edu	0.000	501	INTEL	LINUX	621772	Claimed	Busy
cruella.cs.wisc.edu	1.000	255	INTEL	LINUX	103419	Claimed	Busy
curds.cs.wisc.edu	0.650	511	INTEL	LINUX	590793	Owner	Idle
cygnet.cs.wisc.edu	1.000	127	INTEL	LINUX	68667	Claimed	Busy
dameedna.cs.wisc.edu	0.990	511	INTEL	LINUX	342808	Claimed	Busy
daphne.cs.wisc.edu	1.000	255	INTEL	SOLARIS26	83516	Claimed	Busy
diamond-joe.cs.wisc.edu	0.300	511	INTEL	LINUX	332888	Claimed	Busy
diana.cs.wisc.edu	1.540	511	INTEL	LINUX	327990	Owner	Idle
dingo.cs.wisc.edu	0.990	511	INTEL	LINUX	352764	Claimed	Busy
dogberry.cs.wisc.edu	0.990	255	INTEL	LINUX	275698	Claimed	Busy
dov.cs.wisc.edu	1.000	255	INTEL	SOLARIS26	83577	Claimed	Busy
dr-hibbert.cs.wisc.edu	1.000	500	INTEL	LINUX	344627	Claimed	Busy
durden.cs.wisc.edu	1.030	255	INTEL	LINUX	298551	Claimed	Busy
eagles.cs.wisc.edu	0.990	511	INTEL	LINUX	317287	Claimed	Busy
elbow.cs.wisc.edu	0.990	255	INTEL	LINUX	250461	Claimed	Busy
elroy.cs.wisc.edu	0.990	511	INTEL	LINUX	342936	Claimed	Busy
environ.stat.wisc.edu	0.000	254	INTEL	LINUX	204236	Owner	Idle
espo.cs.wisc.edu	1.010	511	INTEL	LINUX	101150	Claimed	Busy
euler.cs.wisc.edu	1.230	249	INTEL	LINUX	563968	Claimed	Busy
everest.cs.wisc.edu	0.990	255	INTEL	LINUX	215683	Claimed	Busy
falstaff.cs.wisc.edu	0.990	255	INTEL	LINUX	246920	Claimed	Busy

Fig. 1: Default table showing all classads for Intel machines

Data Formatting

All initial data is displayed in a default configuration. In this configuration, all reals are displayed to three decimal places and integers and strings are displayed as is. The user is then allowed to change the way some data are displayed. Strings do not allow any formatting. They are displayed as is and the user has no control over that. Integers on the other hand are highly flexible since they are used to represent not only what would be a natural number in mathematical terms, but also dates and times such as the number of seconds elapsed since a machine went idle. The integers that actually represent dates or times are shown in the table below.

Attribute Name	Default Values Measured In
ClockDay	Days
ClockMin	Minutes
ConsoleIdle	Seconds
CpuBusyTime	Seconds
EndDownTime	Seconds
EnteredCurrentActivity	Seconds
EnteredCurrentState	Seconds
JobStart	Seconds
KeyboardIdle	Seconds
LastBenchmark	Seconds
LastHeardFrom	Seconds
LastPeriodicCheckpoint	seconds

These (dates) can be displayed as is (number of days, seconds, or milliseconds), or can be formatted in minutes, or hours, or days. As for displaying reals, the user can choose how many decimal places to display, whether to use scientific notation, or whether to use percentages if appropriate. There are actually few attributes whose type is real. These are: CondorLoadAvg, CurrentRank, LoadAvg, Rank, TotalCondorLoadAvg, and TotalLoadAvg. For both dates and reals, the application opens a modal window editor where the data display is formatted at will.

Finally, users are allowed to do sorting on a chosen attribute value. For instance, they can choose to arrange the view in ascending order of available memory.

3.2 User Input

Since all the data being gathered cannot be changed and since the application is primarily a graphical user interface, the only input that is allowed from users is how the data can be displayed. This will be provided exclusively via menus and mouse events, and perhaps some mnemonic key strokes to stand in for some specific mouse events. Configuration files in the user's home directory, if they exist, are read in during application load time and override the default configuration provided by the application.

3.3 Output

All output will be reflected by changes to the interface displayed on the user's console. However, when the user selects to save a customized view, this will be written locally to the user's home directory as `classadview.config` and read in during the next time the application is loaded. In addition, some statistics may be gathered and displayed in a tabular or graph format. These statistics will include metrics such as averages and totals. Figure 2 shows a screenshot of the application's default output after saving `classadview.config` in the user's home directory.

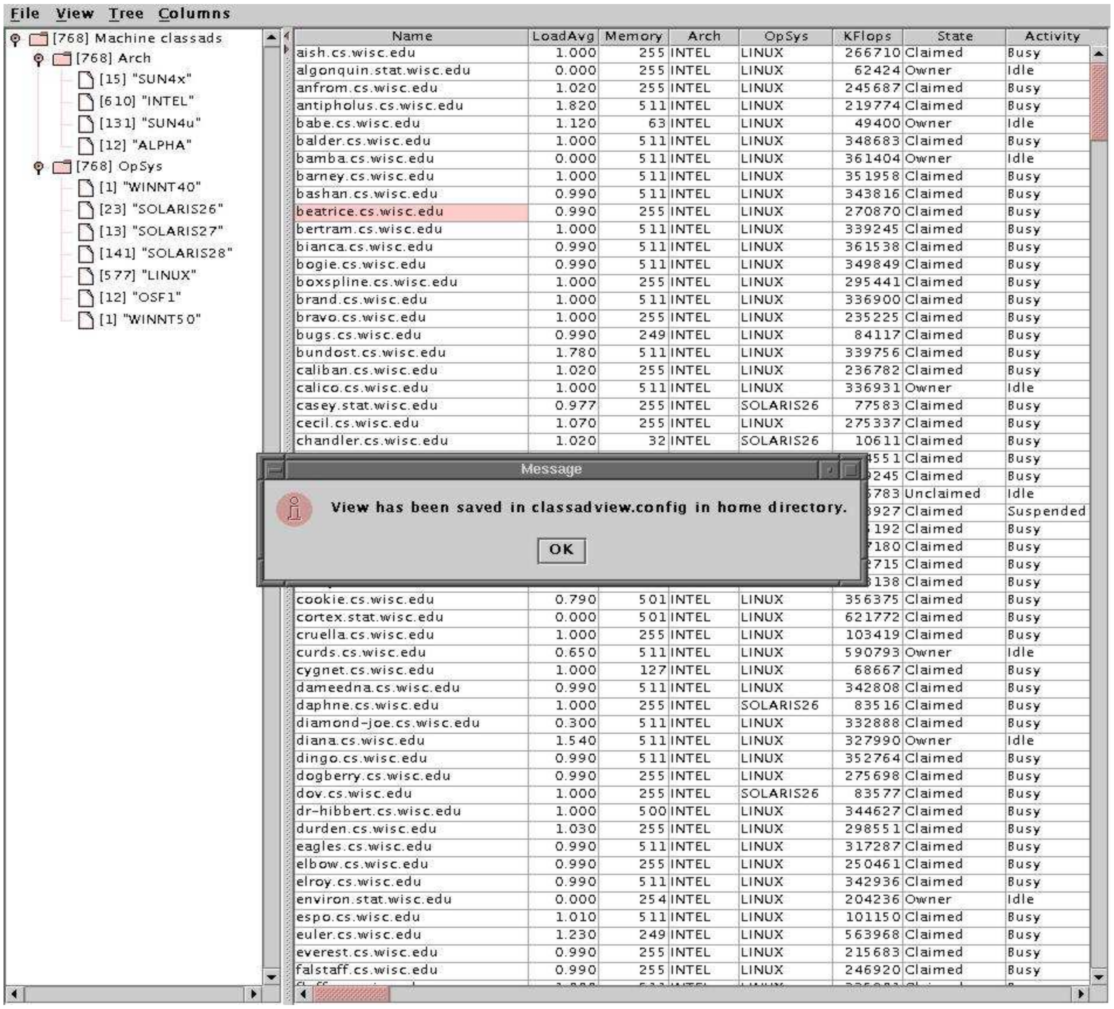


Fig. 2: Output after saving preferences in home directory

Figure 3 shows a screenshot of the Add menu item deployed for adding new nodes to the tree view.

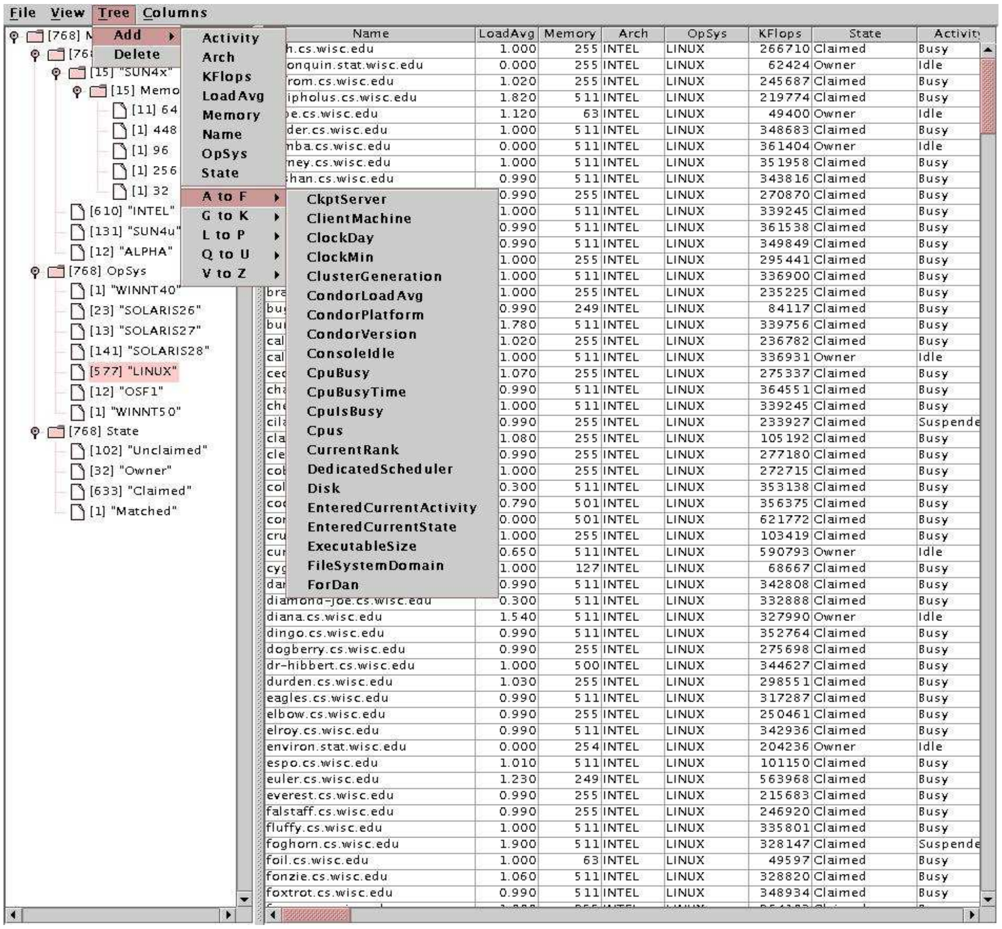


Fig. 3: Adding a new node to the tree

3.4 System Database/File Structure

At load time, the application accesses any saved configuration file in the user's home directory and reads it instead of the default configuration. This file is created upon the user's selecting to save a customization as explained above. The application maintains all of its data in memory while it is running, accessing local disk only to write out the 'classadview.config' file or to read it in. A menu item that allows to write the classads to file has been provide under the File menu.

3.5 Limitations

The application performs all of its functions on the copy of classads that it has in memory. As a result, the data presented may be stale since the last reading. The data is only correct as of the time of the last reading from the collector. However, the refresh feature allows users to direct the application to fetch the latest classads from the collector. The application does not allow any changes to the data retrieved from the collector. It only changes how that data is displayed.

4 System Performance

4.1 Efficiency

Considering the fact that hundreds of workstations exist in the Condor pool and that their number is growing and considering that each workstation has scores of attributes, efficiency is a concern. The application and all of the other components it communicates with are expected to perform in a reasonable manner. The browser should be at least as efficient as a commercial operating system browser, such as the Windows file explorer.

In the early stages of the development, significant memory leaks by the application were observed. The problem reached such a height that it was not uncommon for the application to run out of memory. After proper investigation, it was found that the requirements expressions (essentially in-memory strings) where being duplicated every time the refresh feature was invoked. Changing the implementation of these string representations from arrays to linked lists, along with using references to single copies of data reduced significantly the magnitude of the problem. This experience only reinforced the importance of choosing the appropriate data structure to hold data.

The application requires about 25 MB of available memory to run. Archived into a jar file, it is about 125 KB in size. The longest operation, the initial reading of classads via the network connection, takes on average 12 seconds, much of which is attribute to network latency. The same measure is true during refreshes. Other than this, the slowest operation, displaying all classads into the table, takes on average 4 seconds.

4.2 Reliability

The application guarantees that the data displayed is in fact the correct values gathered from the classads. This may sound trivial but if care is not taken on manipulating the classads, the application will display erroneous values. A similar issue was encountered during implementation, when it was noticed that the RemoteUser attribute had sometimes values other than ’claimed’. After the proper investigation, a bug in the algorithm that handles search for attribute values was discovered. Once the problem was identified, it took about five lines of code to fix the problem. In addition to data reliability, the application itself is quite reliable since no crashes or exceptions were observed during testing.

4.3 Security

The application provides read-only access to data that is made publicly available through a published Condor interface. All view manipulations are occurring to an in-memory copy of the data. Once the application requests data from the collector and is provided the data, all communications with the back-end are severed.

5 System Design Overview

5.1 Organization

The project’s organization can be decomposed into three major components as follows. The graphical user interface (GUI) that presents the data to the user and accepts input from the user is at the highest level of the application. The data being presented in the GUI is formatted for human readability, though the internal data structure representation of such data may be otherwise. For instance, a number may be presented in the table as a string, whereas its internal representation may be an integer or a double. The GUI is typical and will incorporate menu bars, menu items, trees, window panes, command buttons, and so on, and will be accessed primarily via mouse events such as mouse clicks.

The second component of the application is a set of two external libraries (or packages to borrow from Java terminology) to read and write binary data in a format compatible with the Condor Cedar C++ library and to implement Classads in Java. User input is reformatted in a manner understandable to the layer below, and results coming back from that same layer is formatted for human readability and passed on above to the GUI for display. The Cedar and Classad packages are used during this phase of execution.

Finally, at the lowest level of the organizational hierarchy is the Condor collector, which receives the formatted input passed down by the libraries. This module accesses core Condor functionality and returns the output of its execution back up to the application. Additional documentation for this component in particular and the Condor system in general can be found in [5].

5.2 Required Support Software

As already briefly mentioned above, the application makes use of other modules within the Condor distributed system in order to function adequately. These modules include the Cedar and Classad packages, and the collector. Furthermore, network connections are established during querying between the application and the Condor server. As such, communications must be up and running and maintained during this phase of the execution. In the event that communication cannot be established between the application and the Condor server, execution is still possible but only the local in-memory data of the last call to the collector would be available to the user. Since this data is in-memory and might not have been saved to local disk, a complete shut down of the application could mean loss of the aforementioned data. From that point onward, communication must be re-established with the Condor server in order to obtain new data.

5.3 Implementation Language and Hardware Configuration

All of the implementation of the project was carried in the Java programming language. Java was chosen primarily because of the following reasons.

- Ease of understanding and use
- Good support for GUI development
- Freely available
- Portability

The application performs reasonably well from the user’s point of view and no performance degradation was observed during system testing. Occasionally, if the user selected to display all attribute values in the table (rare case), this function took about 4 seconds.

The project was primarily carried out on a Linux workstation, running Redhat 6.2, with 512 MB of available memory, and a 1.4 Ghz AMD processor. The project was also tested on NT and Solaris platforms in the CS lab. The results are summarized in the table below.

Platform	CPU	Clock Speed	Available Mem	Used Mem	Load Time	Refresh Time
Linux	AMD	1.4 Ghz	512 MB	24 MB	8 secs	9 secs
NT4	HPVectra	400 Mhz	64 MB	24 MB	10 secs	10 secs
NT2000	P III	700 Mhz	256 MB	24 MB	9 secs	10 secs
Solaris28	Sparc	550 Mhz	256 MB	28 MB	14 secs	12 secs

6 System Data Structure Specification

The design of the application called for using efficient data structures to store the main data at hand, namely: the classads, the attributes, their values, and their types. The classads gathered from the collector are stored in a vector of Record Expressions. These are the expressions that are searched during an attribute value lookup. The value, if it exists, is anything after the '=' sign and up to the next semicolon (;). The attribute types are stored in a hash table and keyed by the attribute names. The classad expressions are displayed in a table, with each attribute name assigned a column. The application handles and manipulates data in two ways. There is first the internal representation of each data type: classad integers are represented as Java int's, classad reals as Java doubles, UNDEF, ERROR, strings, and boolean as Java strings. The other aspect of data display is that once the application is ready to display to the table or the tree, everything becomes a string. This makes it easy to customize data formatting for example.

7 Module Specification

In this section, we describe the complete specification of the different classes used to implement the application. The word "module" is loosely used here and should be understood to refer to Java classes. Parameters to methods calls have been left out and can be obtained directly from the source code.

7.1 DataNode

This object serves as the base class for nodes in the underlying tree. Classes that inherit from DataNode are AttNode, ValueNode, and AttValueNode. Only one object in a tree is of type DataNode - the root. All other nodes in the tree are an instance of a derived class. The root node holds all the classads gathered by the collector and is the most general classad container in the tree view as mentioned in section 3.1. This class has two methods of importance:

`determineChildren():` for attribute nodes, the tree structure is used to determine the children. For value nodes, the classad vector is used instead. Thus, the method's functionality is implemented in derived classes AttNode and ValueNode. The children are stored in a hash table named 'children'.

`addToViewedTree():` this method will only be explicitly called on the root of the tree. A corresponding node is added to the JTree object that serves as the GUI for the tree. Then the method is called recursively on all the children of the current node. As a result, calling `addToViewedTree()` on the root of the underlying tree creates the entire JTree.

7.2 AttNode

This subclass of DataNode stores a node that corresponds to an attribute (for example, 'OpSys'). The parent of an AttNode will be a DataNode (the root node), a ValueNode, or an AttValueNode. This class overrides the following method from its parent, DataNode:

`determineChildren():` based on the classads stored at an AttNode, this method finds all the possible values of the attribute. For each of these values, a new ValueNode is created, and becomes one of the children of the AttNode.

7.3 ValueNode

This subclass of DataNode stores a node that corresponds to a particular value for an attribute in a classad (for example, 'SOLARIS28'). The parent of any ValueNode object is always an AttNode. For the above example, the parent would be the 'OpSys' AttNode. This class overrides the following method from its parent, DataNode:

`determineChildren():` enumerate through parent's classads, look up on parent's attribute, compare to name and if there is a match, add a reference to the vector of classads.

7.4 AttValueNode

This class has not been implemented yet. It is meant to be used for a special case, in which an AttNode would only have one ValueNode for a child (i.e. all classads at the AttNode have the same value for that attribute). The idea is that instead of using both an AttNode and a ValueNode (which is redundant), use an instance of AttValueNode that stores both the name and value of the attribute.

7.5 ClassadData

This class reads the classads from the collector, and performs several other useful functions such as gathering all the classad attributes, parsing the classads, determining the attribute types, reading the classads from a file (if necessary), and writing the classads to a file (if necessary). All the data structures and methods in this class are static so the class need not be instantiated to access these. It has the following important data structures:

'_myClassAdExprs': a vector of classads
'_attribs': a hash table of attribute/value pairs
'_attribTypes': a hash table of attribute/attribute type pairs
'sortedAttribs': a linked list of the alphabetically sorted attributes

It also has the following methods:

`getServerName():` Return the name of the server from which data was read.

`getClassAd():` this method is used to get the complete value of a classad given the name of the machine you want the classad from. Return the string representation of the particular classad.

`getAllAttributes():` get all attributes and store them in the _attribs hash table in lower case to avoid possible capitalization problems.

`determineAttributeTypes():` for each attribute, determine its type and store it as an entry keyed by the attribute name in the _attribTypes hash table.

`readClassAds():` read the classads from the given server. Assign a vector the classads, null if there was an error in reading the adds. The query does not need a where clause since it is reading all the adds available.

`sendQuery():` send the ad to the server and return a vector of classad expressions.

`readDataFromFile():` read the classads from a file, if a file name is supplied instead of a Condor server name. The file is a text file of classad expressions. It returns a vector of classad expressions.

`writeDataToFile():` writes the classads to local disk. The file is a text file of classad expressions.

7.6 DynamicTree

This class is used to build a dynamic tree. A dynamic tree is by definition a tree that can be updated by adding or deleting its children, be they leaves or subtrees of their own. The changes are immediately visible. This class has the following important methods:

`removeNode():` remove a selected node from the tree view.

`addNode():` add an attribute to the tree view as a new DefaultMutableTreeNode.

7.7 ClassadTable

This class is used to build the table that holds the values of the attributes being selected from the tree view. It manages columns, rows, event handling such as mouse clicks for sorting the data, opening new windows for data customization, adding and/or deleting columns from the table, and rearranging the table header. It has the following important methods:

`createDefaultTable():` create the default table consisting of the eight default attributes Name, LoadAvg, Memory, Arch, OpSys, KFlops, State, and Activity.

`fillClassadTable():` populate the classad table upon selection of a tree node from the tree view.

It has the following inner classes:
`ColumnKeeper{}:` keep track of column addition and removal from the table

`InitialData{}:` handle classads for the default eight attributes

`ColumnData{}:` class to hand a column’s name, width, and alignment

`ClassadTableData{}:` manage all the classads, the row count, the column count, getting values from a certain cell, registers column listeners and column movement listeners.

`ClassadComparator{}:` for comparing column values and allowing for sorting

7.8 GuiManager

The work-horse class for the classad viewer application. This class is the primary class that is loaded and executed and will be calling methods from the other classes to perform specific tasks. It uses the ClassadData class to read classads from the collector, creates an underlying tree to store classad references based on this tree, and creates the JTree object that serves as the GUI. It has the following important methods:

`GuiManager():` this is the constructor. It connects to the server to read classads, creates the tree nodes, draws the tree, and creates the default table.

`createNodes():` create the underlying tree to store classad information.

`drawTree():` based on the underlying DataNode tree, create a JTree with which the user can perform selection.

`createMenuBar():` create the application's menu bar.

`refreshTree():` get latest classads from the collector.

`savePreferences():` save the preferences that the user has selected by writing them to disk in the current directory.

`loadPreferences():` load any preferences for the tree view that may have been saved in previous sessions of the program.

8 Future Work

The first task of the present project consisted in providing a basic appearance for the tool and in allowing customization of this appearance. The next task will consist in providing data formatting. Each column's data in the table view could be formatted to display its values in a certain way, depending on the type of the values being displayed. For example, LoadAvg could be formatted to be displayed in percentage instead of a real or TotalVirtualMemory in GB instead of MB. This will be the subject of the project's next phase.

9 Conclusion

From taking over the project in its early infancy, we have been able to add more functionality to the basic appearance over time. From displaying a simple default tree, the project moved on to allow for node addition, node removal, refreshing the tree data, displaying the classads in a table, adding or removing columns from that table and saving customized views. Many more features that allow the user to manipulate appearance were added. These include sorting values in a column, changing column width, and reordering columns. We learned that successfully designing and implementing a GUI requires constant feedback between implementer and user. We were able to implement all essential basic appearance functionalities.

References

[1] Peter van der Linden. *Just Java 2*. The Sun Microsystems Press Java Series, Forth Edition.

[2] Mary Campione and Kathy Walrath. *The JFC Swing Tutorial*. Addison-Wesley, November 2000.

[3] Mathew Robinson and Pavel Vorobiev. *Swing*. Manning. May 2000.

[4] Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification. WWW, 2001. http://java.sun.com/products/jdk/1.2/docs/api/index.html

[5] The Condor Project Home Page. WWW, 2001. http://www.cs.wisc.edu/condor/