Flexible Collection and Exploration of Condor Monitoring Data

D. Koelewijn

Parallel and Distributed Systems Group
Department of Mathematics and Computer Science
Delft University of Technology

January 1998

Supervisors:

Prof. Dr. Ir. H.J. Sips (TU Delft)
Dr. R. van Dantzig (NIKHEF)
Ir. Dr. D.H.J. Epema (TU Delft)

Preface

This report is the result of a six-month graduation term at the Parallel and Distributed Systems Group of the Department of Mathematics and Computer Science of the Delft University of Technology (DUT). This project was performed within the Computer Technology group (CT) of the Dutch National Institute for Nuclear and High-Energy Physics (NIKHEF), under the supervision of R. van Dantzig (NIKHEF) and D.H.J. Epema (DUT). I would like to thank R. Boontjes and W. Heubers of the CT group for their help during my stay at NIKHEF. I am grateful to M. Livny and R.K. Wenger of the Condor and DEVise teams at Wisconsin-Madison for providing enthusiastic support. P. Anita deserves thanks for helping me out in the Delft Condor pool. And finally, I would like to thank D. Epema and R. van Dantzig for their supervision of my graduation term, for their guidance and encouragements regarding my work, and for their comments on the drafts of this report.

Denis Koelewijn,
Amsterdam,
January 1998

Abstract

Condor is a distributed batch computing system which allows users to execute batch processes, called Condor jobs, on unused computers in a network of workstations. Condor simulates the local environment of the submission machine to its jobs; as a result, a Condor job can access files as if it executes on the submission machine. Condor creates a checkpoint of a running job when a machine is no longer available, that is, when the owner of the machine on which the jobs is executing, starts using the machine again.

CondorView is a monitoring utility for performance evaluation of Condor. CondorView collects data from a Condor pool, and it visualizes the current and past utilization, and a number of other metrics of the Condor pool.

In my master's project, CondorView has been redesigned and re-implemented in two separate parts, one for monitoring and one for exploration of the monitoring data. The entire monitoring part is now designed using object-oriented methods, and is prepared for Condor version 6. When new information becomes available from Condor or from monitoring tools, the monitoring part can easily be extended. For the exploration, CondorView now uses the visualization tool DEVise, which has powerful exploration primitives. Instead of fixed graphs, we can now create complex presentations using exploration modules. Three such modules have been defined and implemented, one for the current status of a pool, one for the averaged history of a Condor pool per machine type, and one for the history per machine. Other modules can easily be added later on. The use of the new monitoring structure has been demonstrated in a number of exploration examples.

Contents

List of Figures

Chapter 1

Introduction

With the shift of computer power from centralized mainframes and supercomputers to desktop computers, users have acquired more control over their computers. At the same time, more and more computer resources (processors, memory, disks, etc.) are left unused. This has two causes: Users are considered the owners and sole users of the resources of the computer on their desks, and secondly, with the distribution of computers, computer resources also become distributed, and thus become more difficult to be put to general use. Distributed computing systems offer a single 'metacomputer', which hides the details of the actual distributed computers. With a metacomputer, a user can gain access to the distributed resources as if they are part of the local machine. Monitoring the performance of such metacomputers is important for research purposes as well as for system maintenance. This report describes a new, flexible data collection and exploration architecture for the distributed batch computing system Condor[1]. The new architecture, created for this master's project, allows more extensive monitoring and more flexible visualization of the collected monitoring data than was available before.

Condor [3] is one of the earliest distributed batch computing systems. The objective of Condor is to make available otherwise unused computer cycles in a network of workstations. Measurements on a network of workstations at the Computer Sciences department of the University of Wisconsin-Madison have shown workstations to be idle 70% of the time [15, 16]. Even during the business hours, 50% of the workstations were idle. These observations and the notion that a group of so called 'frustrated' users exists, always with a need of more processing power, form the most important reasons for developing Condor. Condor aims at solving the wait-while-idle problem, in which a number of users need more processing power than they have available on their desks, while other computers in the local network are idle. Users can submit batch jobs to Condor, which executes the jobs as soon as a suitable machine becomes idle. A collection of machines available to Condor is called a Condor pool. Several Condor pools can be connected together to form a Condor flock. In a Condor flock, waiting Condor jobs are transparently migrated to other pools when computers of other pools become available. The advantages of Condor flocking have been demonstrated in a pilot project, in which nine universities and institutes from Europe and the United States have formed one Condor flock in the past [5].

[1]The Condor home page is located at `http://www.cs.wisc.edu/~condor`. Information on the work on Condor done at the Delft University of Technology and the NIKHEF can be found at `http://pds.twi.tudelft.nl/~condor`

Other developments in wide-area distributed systems are Globe [10], Networks of Workstations (NOW) [1], Metacomputer Online (MOL) [18], Globus [6], Legion [7] and the Polder Metacomputing Initiative[2].
Globe is a project of the Vrije Universiteit of Amsterdam. It is planned to be a transparent, object-oriented distributed operating system. Resources in Globe are represented by objects, which can be shared among processes. Each object can be distributed over several physical locations and it can be addressed independent of its location. However, Globe is still in the early stages, no actual implementation is available yet.
The Berkeley NOW project tries to integrate all the computers in a building to satisfy the needs of both desktop computing and applications that need a hundredfold more computing resources than found on any single machine within the building. This is achieved by providing three new functions: improving virtual memory and file system performance by using the aggregate DRAM of a NOW as a giant cache for disks; achieving cheap, highly available and scalable file storage by using redundant arrays of workstation disk; and finally, using multiple CPUs for parallel computing. These functions make a NOW system more than simply a bunch of machines on a fast network.
The MOL project concentrates on integrating existing modules, such as programming environments (e.g., the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI)), resource management and access systems (e.g., Codine and the Network Queuing system (NQS)), and supporting tools (e.g., graphical user interfaces). Integration is achieved through generic interfaces defined for the effective interaction between the modules.
The goal of the Globus project is to provide a low-level infrastructure that can be used to construct portable, high-performance implementations of high-level services like PVM and MPI. This low-level infrastructure should reduce the complexity and improve the quality of the high-level services. The infrastructure also provides solutions to the unpredictable structure of a metacomputer.
Legion aims at building a (world-wide) virtual computer, somewhat like the WWW, transparently consisting of millions of hosts like workstations, personal computers, supercomputers, and non-traditional devices like TVs and toasters, all connected with high-speed links.
The objective of the "Polder" Metacomputing Initiative is to build a distributed metacomputer using existing distributed batch and parallel systems including Condor.

To investigate whether computer resources are being used efficiently, a performance analysis has to be done. During a performance analysis, bottlenecks may show up, which prohibit better performance of the system. Extensive monitoring of the system under study is needed to obtain insight into its performance. To simplify the task of evaluating the system's performance, an artificial workload can be used. This workload will generate a predictable system load during the monitoring. As part of an earlier study project, the Condor Artificial Workload Generator (CAWG) [11] has been built.

Monitoring in Condor is performed with CondorView. This tool collects information on Condor, and it visualizes Condor's current and past utilization. CondorView is also capable of showing the effects of the flocking mechanism, e.g., the amount of work exported to and imported from other pools. CondorView was originally developed by the Condor team at Wisconsin, and it has recently been extended by van der Star of the Delft University of Technology [21].

[2] `http://www.wins.uva.nl/projects/polder/`

CondorView is not complete, there are still more aspects of Condor that we would like to monitor. Already, a number of monitoring tools exist that still await integration into Condor, for example tools for monitoring job I/O and network bandwidth [21, 12]. Extensive monitoring generates large amounts of data. To obtain insight into these data, flexible user-controlled data selection, analysis, presentation and visualization should be possible. To include these facilities in CondorView, we have decided to use the already existing tool DEVise [13, 14] (DEVise stands for "Data Exploration and Visualization"), which is in an advanced state of development at the University of Wisconsin-Madison. With DEVise, large distributed data sets can be explored and visualized. Various — possibly remotely accessible — data sets can be visualized concurrently, and several users can collaborate on the same visualization at the same time.

The purpose of the work described in this master's thesis was to redesign the Condor monitoring architecture to enable more extensive monitoring and more flexible exploration of the collected data. Besides the standard monitoring tools, additional monitoring tools can now be straightforwardly included. All data collected by CondorView and the additional monitoring tools are stored on disk — possibly distributed worldwide — and accessed by DEVise on demand. Using the interactive exploration and visualization capabilities of DEVise, users can get a good grasp on these data. Several standard presentations have been made with DEVise for a quick overview, as well as for an in-depth analysis of the data. The new CondorView Server has been tested in the Condor pool at NIKHEF in Amsterdam and in the pool at Delft University, together forming a small Condor flock. The CondorView Client has been used to explore monitoring data of the Condor pools at NIKHEF, Wisconsin-Madison and Bologna.

This report is organized as follows. Chapter 2 presents a short introduction to Condor and describes the previous monitoring structure. The chapter continues with a list of functions which the monitoring structure should provide, and the new monitoring structure according to these functions. Chapter 3 provides an introduction to DEVise and an example that explores the possibilities of DEVise. Chapter 4 treats the design and implementation of the new CondorView Server for the new monitoring structure. Chapter 5 deals with the CondorView Client. Exploration examples created with the new monitoring structure are presented in Chapter 6. Chapter 7 finishes this report with a summary and conclusions. Technical details can be found in the appendix.

Chapter 2

Condor Monitoring

A performance analysis of Condor requires adequate monitoring. The monitoring allows us to assess how efficiently Condor processes its jobs, and whether there are specific bottlenecks that prohibit better performance. In this chapter, we present the status of Condor monitoring before this master's project started. Next, we discuss the functions of a new monitoring structure, which allows more extensive monitoring than before. The chapter finishes with a global design of the new structure.

2.1 Introduction to Condor

On each machine in a pool a number of Condor daemons run. These daemons watch "their" machine, for example whether it is idle, and whether it has Condor jobs to execute. Such information is sent to the *Central Manager* (CM), which is hosted on one of the machines in the pool. The CM matches idle machines and Condor jobs. Once it makes a match, the CM notifies the machine where the matched job is submitted. A daemon on the submission machine then contacts the idle machine to inquire whether the machine still is idle. When this is the case, the jobs is transferred to that machine and subsequently executed. The information collected by each daemon and sent to the CM can also be sent to an external monitoring program.

Condor has a number of features that are important to be monitored. One feature is *Remote System Calls* (RSC). The system calls made by Condor jobs are sent back to the submission machine. This is done for two reasons: a) the Condor job cannot access files on the local hard disk; and b) the local environment of the Condor job is identical to that of the submitting user, so files can be found with the same path as on the submitting machine, for instance. However, the system calls and their results have to be sent over the network. Therefore, the network performance between the two machines can be very important, depending on the number of system calls.

Another feature of Condor is *checkpointing*. When the owner of a machine that is running a Condor job starts using the machine again, Condor makes a checkpoint of the job. The job can continue on another machine with the checkpoint file. Checkpoint files generally are quite large, in the order of several (tens of) megabytes, and they have to be sent over the network.

Therefore, again, the performance of the interconnecting network is very important. This is especially true for large pools (hundreds of machines).

2.2 Previous Condor Monitoring

A simple model of Condor monitoring is pictured in Figure 2.1. Users offer a workload to Condor. Condor is continuously identifying the idle machines in the pool and assigning parts of the workload to these machines. During its operation, Condor collects information on the machines and the jobs in the Condor pool. This information is used by Condor internally, but part of it can also be sent to a monitor. Condor version 5 sends only the information on the status of machines to an external monitor. Since we want to monitor more than that, we use additional tools that supply extra measurements to the monitor. Using all acquired monitoring data, we can fine-tune certain scheduling aspects in Condor. In the following sections we discuss each of the parts of Figure 2.1.

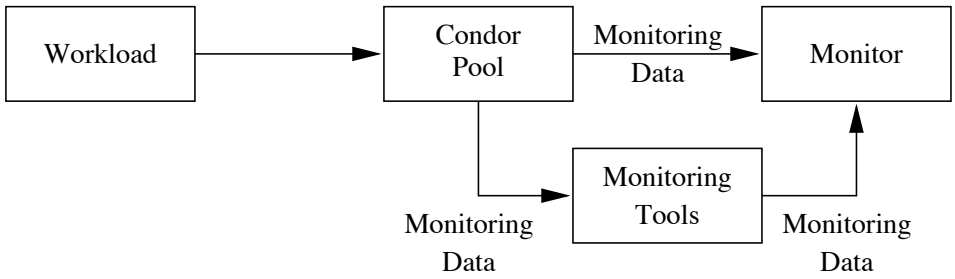


Figure 2.1: A model of the previous Condor monitoring structure.

2.2.1 CondorView Version 3.1

The monitor and visualization tool used at the start of this master's project is CondorView version 3.1. It collects the information that the Condor daemons send it, and then stores all these data in main memory. At the end of each hour, the data of the last hour are averaged and also stored in main memory, in a table with enough space for data of one full year. So, CondorView has information on both the current status of a pool and the history of a pool. One of the most recent additions to CondorView is the B-tree database designed by van der Star [21]. This database was designed to store all collected information on disk in a efficient way and to retrieve these data later according to flexible selections.

CondorView graphically presents certain aspects of the collected data for both the current status and the history of a pool. A presentation of the current status of a pool show the speeds of all machines (in KFLOPS or in MIPS), sorted on their status. Such a presentation displays how many machines are used by Condor and how many are used by their owners at a certain moment. Moreover, this presentation identifies the available but unused machines and the speeds of the used machines. A presentation of the history of a pool shows the same data in a time dependent way for a selected period in the past. This presentation can also show the accumulated amount of computer cycles obtained by all the Condor jobs. It is also

possible to split this graph by submitting user or by submitting pool, if the pool is part of a flock.

CondorView version 3.1 has been designed as a client/server architecture. The server part collects and stores all data, and it holds a number of scripts to visualize the data. The client part connects to a CondorView server to acquire the data and the visualization scripts. These are subsequently executed by the client. The client can actually connect to any CondorView server on the internet, and thus present the data of any Condor pool.

2.2.2 Additional Monitoring Tools

In the previous sections, the network performance and the characteristics of Condor jobs were indicated to be significant for Condor's performance. However, these attributes are not intrinsically monitored in Condor. Therefore, extra tools have been designed to acquire additional information on the performance of Condor. The data collected by these tools cannot be presented by the CondorView version described in the previous section.

The performance of a network can be expressed by two parameters: the available bandwidth and the latency. The bandwidth is the amount of data that can be sent between two nodes per second. The latency is the amount of time between the moment the sending node starts sending its first byte, and the moment the receiving node receives that byte. In other words, the latency is the time it takes to travel the path between the two nodes. The available bandwidth at any moment depends on two quantities: the underlying capacity of the path between two computers, and the amount of other traffic competing for links on the same path [4]. According to this definition, we can distinguish between the maximum theoretical bandwidth and the available bandwidth at any moment, which depends also on the competing traffic. Van der Star [21] implemented a Condor monitoring tool that measures the maximum theoretical bandwidth according to [4]. In [12] a monitoring tool is described, which measures the available bandwidth and latency.

Job I/O monitoring, and more general Condor I/O monitoring, can be performed by an extension to Condor developed by van der Star [21]. This extension consists of a number of changes within the Condor software to monitor the number of I/O operations and the amount of data read and written by jobs and machines. These monitoring data can help the Condor scheduler to make an optimal decision in matching jobs with machines. For example, Condor jobs which perform much I/O must not be sent to a fast machine with slow network connections. On the other hand, jobs which perform very little I/O can be sent to machines with slow network connections. Currently, the monitoring data of job I/O are sent to the owner of the job. These data are not used by Condor or CondorView.

2.2.3 Workload Generator

Usually, Condor's workload consists of large calculations (for example, scientific simulations) and of jobs that process large data files (for example, generated during scientific experiments). So, a simple distinction can be made between computation-intensive jobs and I/O-intensive jobs. Often the workload involves many runs of the same execution program. These runs are often identical, they only differ in the input parameters. The results are accumulated in

a collection of files, which may have to be analyzed in essentially identical ways. Therefore, Condor also accepts jobs in clusters; a cluster contains a number of jobs based on the same executable with different arguments and different input and output files.

It is not possible to estimate in advance how much load an arbitrary job generates. Still, we would like to know how efficiently Condor executes its jobs. Therefore, we construct a fake workload, of which we exactly know the characteristics. This is called an artificial workload. An artificial workload allows us to test Condor in a reproductive way for all kinds of job characteristics. Using an artificial workload, we are not limited to the workload actually submitted by the users of a Condor pool.

For automatic construction of artificial workloads for Condor, the Condor Artificial Workload Generator [11] (CAWG) has been designed recently. The CAWG generates a workload according to a workload description file. This file contains an arbitrary number of lines with parameters, which together define the characteristics of the specified workload. For instance, we have parameters for the number of clusters and jobs in each cluster, and for the characteristics of a single job, such as the computing time and the amount of input and output. It is not necessary to specify all parameters for every job in every cluster, because the workload generator can also be instructed to draw some parameter values from a statistical distribution. Details are given in [11].

2.3 Condor Monitoring Functionality

Now that we have introduced the main parts of the Condor monitoring structure, we can decide what their characteristics should be, and which functions they should offer. The monitoring structure should offer a way to conduct measurements, and a way to organize and store the collected data. It should also enable the manipulation of the data, selection of subsets of data, analysis of these subsets, and finally a flexible way to present and visualize the results. A graphical user interface (GUI) should facilitate all these functions. In the following we discuss the functions involved in the handling of the monitoring data.

Collection The first question concerns what we want to measure. Which metrics do we need to get a good insight into the performance of Condor? A lot of work has been done on this subject. Boer [2] and van der Star [20] have described this issue from different viewpoints. Boer defined two types of monitoring reports: user reports and system reports. User reports include information on the turnaround time and speedup ratio of specific jobs; on the other hand, system reports include overall information, such as the arrival rate and throughput of all Condor jobs. Boer also described how these quantities can be measured and calculated. Van der Star defined a number of monitoring goals, and he described which goals can be reached with the currently available data, and for which goals more data are needed. In [11], we discussed the data needed for a performance analysis in general and for Condor in particular.

Condor collects a certain amount of information that can also be sent to the monitor automatically. As previously described, this information is not complete, and therefore we also want to collect information using the additional monitoring tools. These tools should fit in the monitoring structure.

Structuring The collected data need to be organized in an adequate structure. We want to store these data in an independent format, which can be read by different tools without difficult-to-maintain and non-robust transformations. Transparency is offered by storing the data in ASCII tables. Different types of data can be collected in separate tables, which can be linked on date and time, on machine name, user name, etc.

Part of the structuring is also generating summaries of collected data. These summaries can be used instead of complete data sets, when only a global overview of the data is needed. Moreover, the summaries are much smaller than the complete data sets, so exploration is much faster. Once interesting sections in the data are found using the summaries, the original data sets can be used to explore the details.

Storage We have chosen to store data on disk. This is to be preferred to storage in main memory, because stored on disk, no data are lost if the monitor crashes; besides, the data can only easily be exchanged between monitoring tools, when they are stored on disk. However, some mechanism should be available to maintain the data, to make sure they do not grow beyond limits, to archive old data and to delete obsolete data.

Conversion Data conversion embodies the processing of data for which no understanding of the data is necessary. Examples of data conversions are the transformation of data from binary form to ASCII files and compression of data files. These operations are not necessarily initiated by a user.

Selection The amount of collected data can grow rapidly: different monitoring tools and the standard measurements all generate new data every week, day, hour or even every minute, and every measurement can contain large amounts of variables. Even more importantly, for a certain study usually only a small fraction of the collected data (certain metrics) is relevant. The possibility to select a part of the data is therefore crucial. However, selection should be very flexible. It should be possible to select easily on different variables at the same time. For instance, we do not only want to select all data in a certain period, but also data for machines of a particular architecture, or machines with a certain amount of memory.

Analysis Data analysis consists of operations like making histograms, counting occurrences of certain events, making curve fits, etc. These operations can be done once for all data in a standardized way or they can be initiated when needed by a user trying to comprehend a certain aspect of the monitoring data.

Presentation The presentation defines which parameters should be displayed, how many parameters should be shown per graph, and how these parameters are logically connected.

Visualization The visualization defines how the presentation is actually made, using which colors and forms, whether to use linear or logarithmic scales, isometric or contour graphs, etc.

Graphical User Interface The GUI allows the user to easily browse through all the data in an intuitive manner, without being unnecessarily disturbed by presentation and visualization details. The GUI also allows the user to easily manipulate data sets, change the selection, etc.

2.4 Global Design of the New Monitoring Architecture

Section 2.2.1 discussed the structure of CondorView version 3.1. This program has been split into a server and a client part. The server collects, stores and manipulates the data, and it also has scripts to present these data. The client receives the scripts and the data from the server and takes care of the actual visualization by executing these scripts. Users can perform a kind of selection on the data, but the possibilities are severely restricted. No data manipulation and analysis can be performed at the client level, and the presentation and visualization possibilities are fixed in the scripts. Thus little data processing is possible on initiative of the client side.

If we want the user to have more control to actually explore the data sets, we have to reconsider the way the functions described in the previous section have been split. We can logically split these functions in two groups:

1. Data collection, structuring and storage.
2. Data conversion, selection, analysis, presentation and visualization, and the graphical user interface.

The first part collects the desired data, and stores these on disk. These data are complete and anything can be done with them at any moment. Not until the second logical group any real data processing takes place. The first part is a server that runs in the background without user interaction, and the second part is a client that processes and visualizes all collected data. The client is effectively controlled by the CondorView user.

However, we have not actually built the functions of data manipulation, selection, analysis, presentation and visualization in the client part. All these functionalities can be performed by a specialized exploration program. In the new monitoring structure for Condor, the choice has been made to use DEVise. This tool can provide a exploration engine inside the monitoring structure. The CondorView client offers the GUI and interacts with DEVise

The new monitoring structure is pictured in Figure 2.2. The CondorView Server automatically receives data from the Condor pool. For extra monitoring, additional tools can be registered at the CondorView Server, which starts and stops these tools. These monitoring tools gather additional information on Condor's performance, for instance from the Condor utilities and Condor log files. Monitoring data are stored on disk, accessed through DEVise, and visualized according to the instructions DEVise is given through the CondorView Client. The visualization is now controlled by the CondorView Client, instead of the CondorView Server. Users can explore the data with the GUI the CondorView Client offers. The client translates these user interactions into DEVise commands.

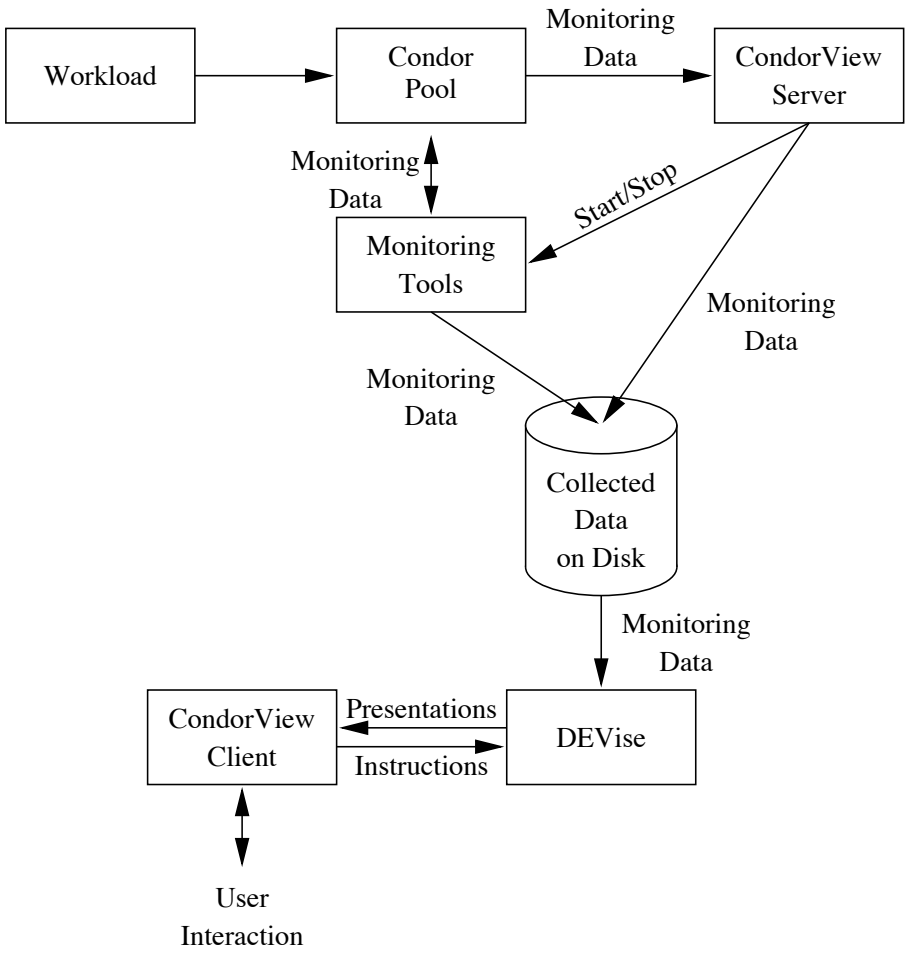


Figure 2.2: A new design of the monitoring structure for Condor.

Chapter 3

Introduction to DEVise

Monitoring Condor tends to generate an enormous amount of data. To get insight into these data, we must have flexible exploration functions. These functions can be developed especially for CondorView, but this is not necessary, because there is nothing special about the data. It is much better to use an already available external exploration program. We have chosen DEVise [13, 14] to do this job, because this program offers flexible data selection, presentation and visualization functions, and because DEVise allows us to create our own graphical user interface on top of it, effectively embedding the capabilities of DEVise in the new monitoring structure. This chapter first presents an introduction to DEVise, and then an example of the capabilities of DEVise.

3.1 DEVise Basics

DEVise visualizations are built up in a number of steps from data files to a collection of interrelated *views*. First we discuss how one view is generated from a data file, and next we describe the possibilities to *coordinate* views.

3.1.1 The DEVise Visualization Model

The visualization model of DEVise is depicted in Figure 3.1. An input data set for DEVise is called a *TData* (for 'tabular data') set. This is a table containing a collection of data records, each composed of the same number and type of attributes. Each attribute represents one column in the table. A *scheme* is associated with every TData set. A scheme file contains the name of the scheme, the type of data (ASCII or binary) in TData sets of this type, and a list

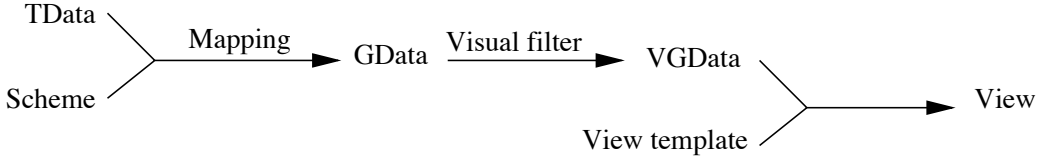


Figure 3.1: The visualization model of DEVise.

of attributes in the data set. An attribute has a name, a domain (integer, float, string, etc.) and optional extra parameters, for instance, the upper and lower limits of integer attributes or the maximal length of a string. TData sets with identical structure can use the same scheme. A visualization made for one TData sets can be used for all TData sets associated with the same scheme.

TData is visualized by creating a mapping, which is a function of the attributes described in the scheme to a graphical symbol with graphical attributes, e.g., coordinates and colour. The mapping creates a collection of graphical symbols, called the *GData* (for 'graphical data'). A mapping can be used for every TData set with the same scheme, because the mapping is associated with the scheme instead of with the TData set.

With a visual filter, a selection on the GData is made using an attribute of the GData. For example, one can select all graphical symbols within a certain range, or with a specific colour. The collection of selected GData is called the *VGData* (for 'visual graphical data'). The VGData form the data-dependent part of a visualization. The data-independent part, also called the *template view*, includes the axis, the title and the background on which the VGData is drawn. A view is constructed by combining these two parts. One or more views can be placed in a window, and one visualization, called a session, can contain several windows.

3.1.2 Coordinating Views

Views can be coordinated to show a number of attributes of the VGData in different views, to show the correlation between two data sets, or to show a data set from different perspectives. In this section we discuss some possibilities to connect views. In the following section we demonstrate how these techniques are used.

Two coordination mechanisms exist, namely *cursors* and *links*. A cursor allows the visual filter of one view, the *source view*, to be highlighted in another view, the *destination view*. If the cursor is moved or resized in the destination view, then the visual filter of the source view is updated, and if the visual filter of the source view is changed, then the cursor in the destination view is adapted. In other words, the destination view with the cursor shows an overview of the data, and the source view zooms in on the data in the part that is highlighted with the cursor in the destination view.

Links establish relations between views. Several types of links exist:

Visual link A visual link is a selection condition that is added to the visual filters associated with each of the linked views. When a visual link connects two views on the x-axis, both views maintain the same range of x-coordinates. So, when the user zooms in on one view, the other view changes accordingly.

Record link A record link connects two views on a set of common attributes. The *slave* view contains only those records that are also displayed (or not displayed) in the *master* view. A record link can be *positive* and *negative*, depending on whether to show in the slave view the records that are displayed, or the records that are not displayed in the master view.

Operator link An operator link is defined on a number of master views and an operator. The master views do not need to use identical or even similar TData sets, but all TData

sets must have at least one attribute in common on which the operator is executed. The result of an operator link is a new TData set. The contents of this TData set may change when the visual filter of any of the master views is altered.

Aggregate link An aggregate link groups the data that is displayed in a view in another view. An aggregate function (such as sum or average) is executed on each group of records in the source view.

The current version of DEVise (1.3.4.1) allows cursors, visual links and record links. These mechanisms are all used in the example in the following section. The other mechanisms are not available in this version of DEVise.

3.2 A DEVise Example

In order to get acquainted with DEVise, we made a presentation of two data sets, generated and visualized by van der Star [21, pages 28–57]. Our presentation, depicted in Figures 3.2 and 3.3, serves as an example of the techniques mentioned in the previous section. First we describe the data sets, and the scheme for each data set. After that, we treat the mappings and coordination mechanisms for this visualization.

3.2.1 Data Sets

We visualized two data sets, both obtained from the Condor pool at NIKHEF. The monitoring frequency used for measuring these data sets was set to three minutes and the measurements continued for 24 hours, starting Monday morning. One data set contains statistics on the pool, the other contains information on each machine in the pool. The data set with statistics on the pool has the following columns: the architecture, the serial number of a measurement and the numbers of machines in states Condor, owner, available and suspended. Some typical lines of this data set are:

```
sun4c 0 0 7 6 0
sun4m 0 11 9 5 2
sun4c 1 0 8 5 0
sun4m 1 11 12 2 2
sun4c 2 0 9 4 0
sun4m 2 12 11 3 1
...
```

The second data set contains information on the individual machines, and has the following columns: the serial number of the measurement, the architecture and the operating system of the machine, the machine name, the pool name of the submission machine of the last executed Condor job on this machine, the owner of the currently running Condor job on this machine, the submission machine of the currently running Condor job, the number of Condor jobs submitted at this machine currently running, the number of seconds the machine is idle, the priority, the timestamp of the last state change, the amount of main memory, the amount of available disk space, the number of seconds the keyboard has not been touched, the number

of processors, the amount of available swap space in KB, the MIPS and the KFLOPS rating, the timestamp of the last time a Condor job was started on this machine, the current state, and the load average. The following lines are typical for this data set.

```
0 sun4c SunOS4.1.2 bora Delft NULL NULL 0 0 0 853933523 12 38716 266 1 28496 12 \
2270 853932545 NoJob 0.000000
1 sun4c SunOS4.1.2 bora Delft NULL NULL 0 0 0 853933523 12 38695 12 1 28496 12 \
2270 853932545 NoJob 0.359375
2 sun4c SunOS4.1.2 bora Delft NULL NULL 0 0 0 853933523 12 38689 0 1 28496 12 \
2270 853932545 NoJob 0.339844
...
0 sun4m SunOS4.1.3 parallax NIKHEF jeroen parallax 14 46 0 854359948 12 792918 \
1318 1 63404 47 6017 854359266 Running 1.019531
1 sun4m SunOS4.1.3 parallax NIKHEF jeroen parallax 14 46 0 854359948 12 792918 \
1439 1 63404 47 6017 854359266 Running 1.015625
2 sun4m SunOS4.1.3 parallax NIKHEF jeroen parallax 14 46 0 854359948 12 792916 \
1683 1 63404 47 6017 854359266 Running 1.000000
...
```

3.2.2 Schemes

The scheme below describes the first data set of the previous section, containing statistics of a pool. Every column, that is every attribute, of the data set receives a name, which is used in a visualization instead of the column number.

```
type menno4 ascii
comment #
separator ' '
attr arch string 10
attr time int hi 479 lo 0
attr condor_mach int hi 40 lo 0
attr owner_mach int hi 40 lo 0
attr avail_mach int hi 40 lo 0
attr susp_mach int hi 40 lo 0
```

The following scheme describes the second data set of the previous section, containing information on each machine in the pool.

```
type menno5 ascii
comment #
separator ' '
attr time int hi 479 lo 0
attr arch string 15
attr OS string 15
attr machine_name string 20
attr pool_name string 20
attr job_owner_name string 20
attr subm_machine_name string 20
attr condor_jobs_running int hi 20 lo 0
attr idle_time int lo 0
```

```
attr prio int lo 0
attr last_state_change_time date
attr total_mem int lo 0
attr disk_avail int lo 0
attr key_idle_time int lo 0
attr cpus int lo 0
attr swap_avail int lo 0
attr MIPS int lo 0
attr KFLOPS int lo 0
attr start_time_job date
attr state string 20
attr loadavg float lo 0
```

Schemes can be made with the DEVise front end, using buttons and menus.

3.2.3 Views

Our visualization is depicted in two figures. Figure 3.2 shows the data set with statistics per machine type in the NIKHEF pool, and Figure 3.3 shows the data set with statistics of each machine in the pool. The visualization uses six windows, five of which are visible in the figures. One window is not visible, because it is only needed for coordination. Later on we describe the function of this last window.

Figure 3.2 contains two windows. The top window titled 'Overview' contains one view, which shows the number of machines used by Condor over all 480 measurements from Monday morning to Tuesday morning. The other window in Figure 3.2, titled 'Attributes per Arch/OS', contains four views. They are, from top to bottom, the number of machines in the state 'Condor', the number of machines in the state 'Owner', the number of machine in the state 'Available' and the number of machines in the state 'Suspended'. In each view, two lines are drawn, one for each architecture. The blue line (or dark gray) is for the `sun4m` machines and the red line (or the somewhat lighter line) is for the `sun4c` machines. In the top and bottom view no red line is visible, because the `sun4c` machines were not used by Condor during the measurements, so the line is obscured by the x-axis.

Figure 3.3 shows three windows. The view in the window 'All Machines' shows the names of all machines in the pool. One machine can be selected, the attributes of this machine are shown in the views in the other two windows.
The window 'Machine Speed: MIPS vs. KFLOPS' contains two views placed over each other. One view contains the coloured dots, which show the speed of all the machines in the pool. The x and y values of each dot in this view are the MIPS and the KFLOPS rating of each machine, respectively. The colour of a dot shows the architecture of the depicted machine, blue for `sun4m` and red for `sun4c`, identically to the colours used in the window 'Attributes per Arch/OS'. The other view contains a vector pointing from $(0, 0)$ to the $(\text{MIPS}, \text{KFLOPS})$ combination of the selected machine. So this vector marks the dot corresponding to the selected machine.
The views in window 'Machine Attributes' show, from top to bottom, the number of seconds the keyboard has not been touched, the number of currently running jobs submitted on this machine, the load average of the machine and the state of the machine. Unfortunately, the state is shown as in integer instead of a string, this is a restriction of version 1.3.4.1 of DEVise.

3.2.4 View Coordination

In the view in window 'Overview', a highlighted area is shown, in the range of approximate 50 to 280, which is a cursor. This view is the destination view of a cursor, one of the views in the windows 'Attributes per Arch/OS' or 'Machine Attributes' is the source view. It is not important which of these views is the source view of the cursor, because all the views in these two windows are linked together with a visual link on the x-axis. In other words, if the visible x-range in one of the views in these two windows is changed, then all the other views in these two windows are updated to show the same range, and the cursor in the window 'Overview' is updated as well. In addition, if the cursor in the window 'Overview' is moved, then all the views will be updated to show the same x-range as the cursor.

There is one invisible window containing one view, which is used to link the three windows 'All Machine', 'Machine Speed: MIPS vs. KFLOPS' and 'Machine Attributes' together. This view is the source view of the cursor shown in the window 'All Machines'. All the views in the windows 'Machine Speed: MIPS vs. KFLOPS' and 'Machine Attributes' are linked with this view with a record link. The effect of this record link is that these views merely display attributes of the selected machine. One view is not linked: this is the view in the window 'Machine Speed: MIPS vs. KFLOPS' with a dot for every machine on the location $(\text{MIPS}, \text{KFLOPS})$.

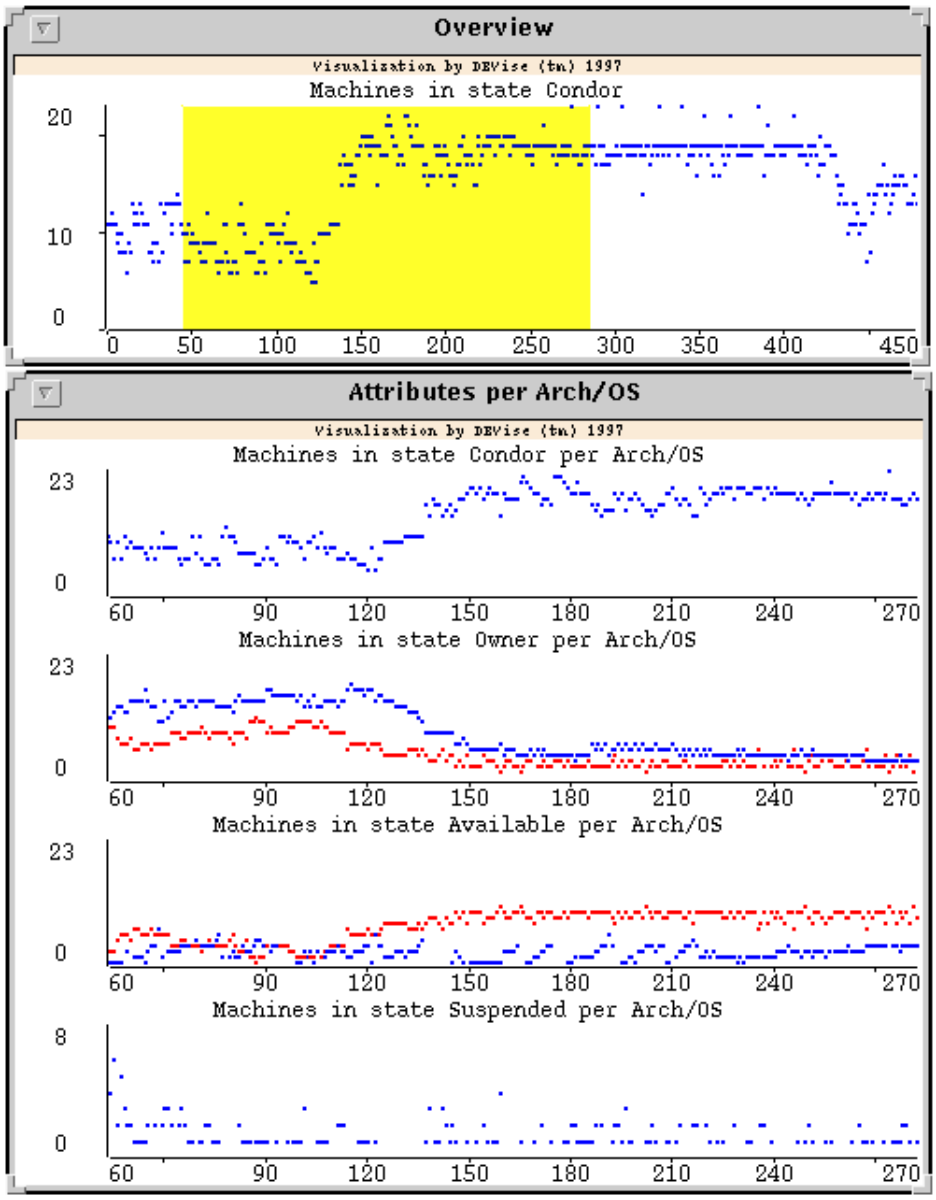


Figure 3.2: The first part of the example. The top window displays the number of machines used by Condor over the complete data set. The bottom window shows four attributes of each measurement for two types of machines and for the cursor in the top window. The view in the top window is the destination view of the cursor, and one of the views in the bottom window is the source of the cursor. All views in the bottom window are linked on the x-axis.

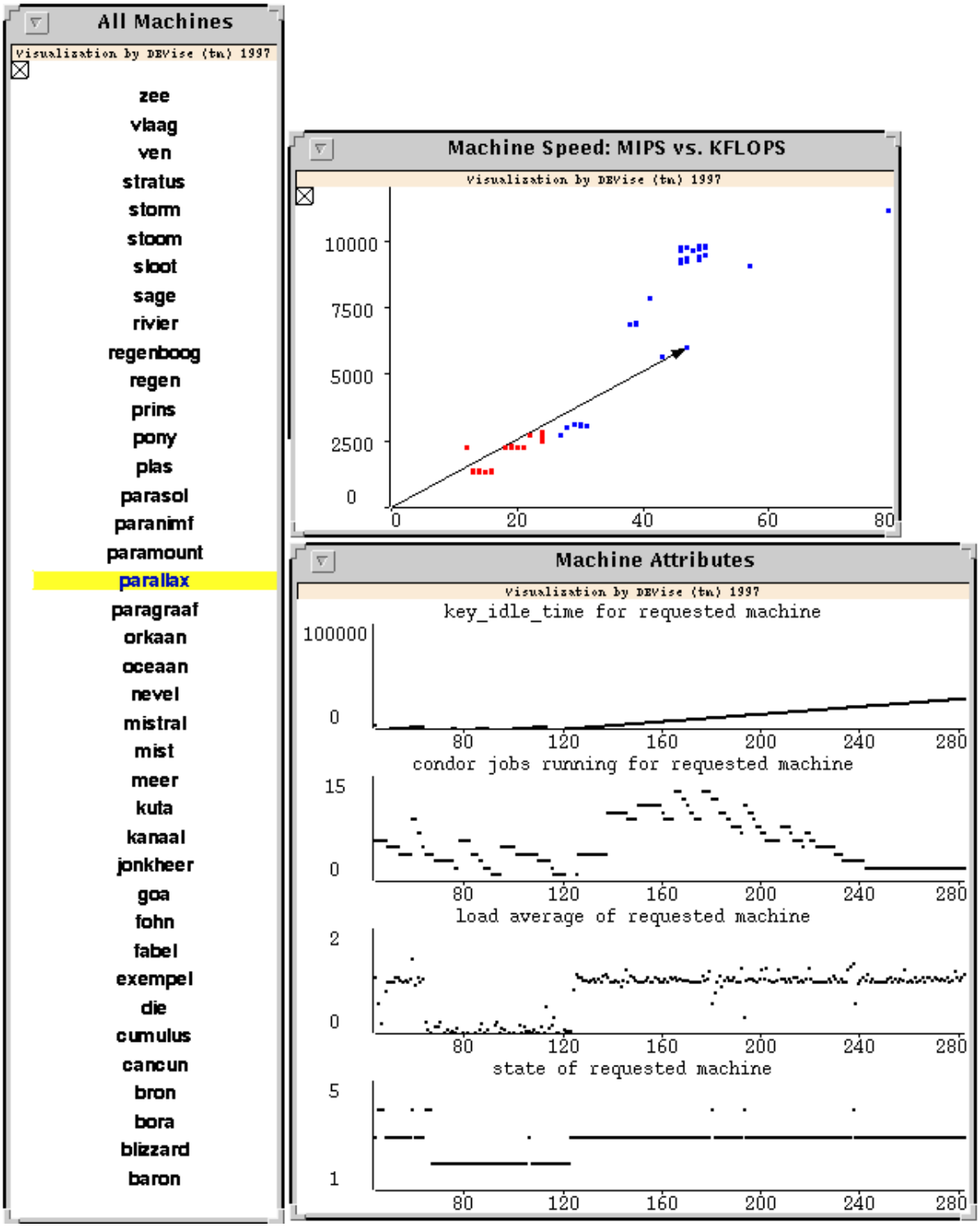


Figure 3.3: The second part of the example. The left window shows the names of all machines in the NIKHEF pool. The top-right window displays the speed of the selected machine in comparison to the other machines in the pool. The bottom-right window pictures four attributes of the measurements of the selected machine over the region selected with the cursor in the top window in Figure 3.2.

Chapter 4

The CondorView Server

The CondorView Server collects the data offered by Condor, and saves these data on disk for visualization by the CondorView Client. The CondorView Server also offers the possibility to start additional monitoring tools for extra monitoring on a regular basis. This chapter treats the design and implementation of the new CondorView Server (version 6.0). A number of classes of Condor version 6 have been used for the new CondorView Server. These classes are described in this chapter. Next, the object model and implementation of the CondorView Server version 6.0 are treated. The CondorView Server version 6.0 cannot be used in parallel with CondorView version 3.1, or without access to the `condor` account. Therefore, a script was created to monitor pools in which the new CondorView Server cannot be used. This chapter finishes with a description of this alternative to the CondorView Server.

4.1 Functionality of the CondorView Server

Section 2.4 mentions that the CondorView Server has three functions:

1. Collection of data from the Condor pool.
2. Structuring the data.
3. Storing the data on disk.

Part of structuring the collected data is generating summaries of the data, because measurements on the NIKHEF pool in Amsterdam have shown that a pool of about 35 machines generates over 3 MB of measurement data per day. This amount of data is not practical if we want to show some statistics of the performance of the pool over the last year, because we will need a too large amount of disk space to store all these data, and the visualization of these data will usually take too much time. The complete original data can be stored for a selected, restricted period of time. These enable the research of the data that are not available in the summaries. We want the following summaries:

Architecture/operating system The number of machines in the states Condor, Available, Owner and Suspended per hour grouped by architecture/operating system.

Machine The percentage of time spent in the states Condor, Owner, Available and Suspended, the number of state switches, the number of running Condor jobs submitted on this machine, the amount of computer cycles consumed by Condor, and a number of other variables per hour grouped by machine.

User The number of Condor jobs waiting, the number of Condor jobs running, the amount of computer cycles consumed by these Condor jobs, and a number of other variables, per hour grouped by submitting user.

Pool The number of Condor jobs waiting, the number of Condor jobs running, the amount of computer cycles consumed by these Condor jobs, and a number of other variables, per hour grouped by Condor pool.

4.2 Design of the CondorView Server

CondorView Server version 6.0 is based on CondorView version 3.1 and the software of Condor. The new CondorView Server, as well as the Condor sources and parts of the old CondorView source, are designed in an object-oriented fashion, which makes it easier to integrate the whole. An important step in object-oriented design is research of the existing classes to identify which classes can be reused. We describe this step in the following section. Subsequently, we treat the object model of the new CondorView Server and we explain this model by describing three scenarios with event traces.

4.2.1 Reuse of Existing Software

Condor Version 6 Classes

Condor contains a great number of classes. A number of classes must be used in the CondorView Server, for instance, the ClassAd. Other classes are not necessarily needed, but facilitate the implementation of the CondorView Server, for instance, the DaemonCore and the CollectorEngine. We now discuss the classes of Condor version 6 that are used in the CondorView Server.

ClassAd A ClassAd is a unit of information for match making between different types of entities, such as the machines and the jobs in a Condor pool. Every ClassAd describes the type of entity it originates from, called the *self object*, and the type of entity that is searched for a match, called the *match type*. Furthermore, a ClassAd contains a number of attributes describing the self object. In addition, a ClassAd contains an expression that must evaluate to true before a match can be made. This expression can contain references to attributes of both the self object and an object of the match type.

DaemonCore The DaemonCore class is used to build Condor daemons. Communication with Condor daemons occurs through TCP and UDP ports. The DaemonCore has methods to create these ports. The TCP and UDP ports are monitored by the `Driver` method, which receives incoming connections and messages. The DaemonCore registers for every type of connection or message a user-defined function. When a known message

or connection is received, the DaemonCore informs the registered function. Connections or messages of a type for which no function is registered, are ignored. Signals are also catched by the DaemonCore and subsequently passed over to the registered user-defined function for that type of signal. The DaemonCore can also register functions to be run repeatedly with a selected frequency. This can be used to clean old data every hour. All these functionalities together make the DaemonCore a very useful class, by which a daemon for Condor can be made very quickly, without much knowledge of the details of TCP and UDP networking or Unix signals.

CollectorEngine The CollectorEngine class is the base of the Condor collector. It contains a HashTable for every ClassAd type. New ClassAds can be added to these HashTables with the `collect` method, old ClassAds can be retrieved with the `lookup` method, and expired ClassAds can be deleted with the `remove` method. Using the `walkHashTable` method, one can execute a chosen function on each ClassAd in one of the HashTables.

The following classes are part of the CollectorEngine class:

CollectorHashTable The CollectorHashTable implements the HashTable abstract datatype. It has methods to insert and remove elements, to iterate over the contents of the table, and to clean the whole table.

HashBucket A HashBucket is the element of storage in the CollectorHashTable. It contains a HashKey on which the element is indexed and a ClassAd as value of the element.

HashKey The HashKey contains the name and the IP-address of the originating machine of the ClassAd and it contains the port number of the connection that was used to transport the ClassAd to the collector.

TimerManager The TimerManager class can be used to register functions to be periodically executed.

CondorView Server Version 3.1 Classes

The following classes are not part of the standard Condor distribution, they are part of the CondorView version 3.1. They are used to store information on the current and past status of the Condor pool.

Arch The Arch class contains a number of statistics per machine type, averaged per hour. This includes the total number of machines of the specific type, the number of machines in the state Available, Condor and Owner, and the sums of the load averages of the machines in each state.

Machine The machine class contains the up-to-date information of each machine from the most recent ClassAd of that machine. This class contains a pointer to a corresponding object of the Arch class, the name, status and load average of the machine, the amount of memory and the amount of free disk space.

These classes are used as basis in the CondorView Server version 6.0, although they are adapted to the new situation in which data are not saved in a BTree, but written directly

to disk in ASCII tables. A number of methods are changed and a new method is added to each class. The method `Roll` of the Arch class was previously needed to discard old data in the arrays that store all collected data. This method is no longer needed. The method `operator «` is added to both classes. When this method is called, these classes will write themselves to a given output stream.

4.2.2 Model of the new CondorView Server

The model of the CondorView Server version 6.0 is shown in Figure 4.1 according to the Object Modeling Technique notation described in [19]. The figure shows the object classes treated in the previous section, and it also shows the new CondorViewServer class. In the Figures 4.2, 4.3, 4.4 and 4.5, event traces of three scenarios are pictured:

1. The startup of the CondorView Server.
2. The arrival of a ClassAd from a Condor daemon.
3. The arrival of a timer signal.

We discuss the function of the CondorViewServer class following these scenarios. Note that the event traces are a summary and simplification of what actually happens.

1. When the CondorView Server is started, it creates a CondorViewServer object. The `Config` method is called to have the object read the global Condor configuration file. After that, the `Driver` method is called to start the monitoring.

   The CondorViewServer schedules a number of periodic tasks with the TimerManager. Subsequently, it announces a number of methods to be called on arrival of ClassAds from Condor using the `Register` method of the DaemonCore. Finally, the CondorViewServer calls the `Driver` method of DaemonCore, after which the DaemonCore waits for incoming ClassAds from Condor.

2. When the DaemonCore receives a ClassAd, the DaemonCore calls one of the registered handlers of the CondorViewServer according to the type of the ClassAd. This handler gives the ClassAd to the CollectorEngine, which stores it in one of the five HashTables (one HashTable for every type of ClassAd). The handlers also calls the `UpdateMachineStatistics` method of the MachineList to update the accumulated statistics of the machine represented in the ClassAd. This method updates the ArchList. If the machine type of the ClassAd is not already available in the ArchList, a new Arch object is created, otherwise the accumulated statistics in the existing Arch object are updated.

3. Functions can be registered in the TimerManager to be executed repeatedly with a fixed frequency. When a function is to be executed, the clock signals the TimerManager, and the TimerManager calls the function to be executed. Currently, two repeating functions are used, one is called every five or ten minutes, the other is called every hour. The first function writes the contents of the MachineList to disk. This represents the current state of all the machines in the Condor pool (see Figure 4.4). The second function writes the

contents of the ArchList to disk. This represents the averaged status of every machine type in the last hour. The ArchList is emptied when all data have been written, to start the averaging anew for the new period (see Figure 4.5).

4.2.3 Implementation

During the implementation we had a number of problems with the Condor classes DaemonCore and CollectorEngine. We applied some bug fixes and changes to these classes to make them work correctly. The details are described in Appendix A.2.

In the pool at NIKHEF, Condor version 5 is installed. With this version only the startd daemons send information to the CondorView Server. Therefore, the collected information of the new CondorView Server is limited to attributes of the machines in the pool. The CondorView Server has no information on the number of Condor jobs submitted, running, suspended, who the owner is of these jobs, etc.

4.3 Alternate CondorView Server

The new CondorView Server cannot always be installed in a Condor pool, for instance when the old CondorView Server cannot be replaced, or when access to the `condor` account is not possible. There are two problems: Condor needs to be informed that it should send ClassAds to the CondorView Server, and Condor can send ClassAds to only one CondorView Server. To overcome these problems, an alternative to the CondorView Server was made. The alternative CondorView Server performs monitoring on Condor using a standard Condor utility, the `condor_status` command, instead of Condor sending ClassAds to the CondorView Server. The output of `condor_status -v` is translated into ASCII tables that can be used by the new CondorView Client. To facilitate monitoring in different pools on any type of machine, the alternate CondorView Server was written as a Perl script. Perl scripts need not be ported to other architecture and operating systems.

The `condor_status` command retrieves the machine contexts for the machines in a pool from the CM. The CM stores in a machine context all data it receives from the startd daemon and the schedd daemon on each machine. Thus the machine context contains information on the configuration of a machine, measurements of the machines, and the number of running Condor jobs submitted on the machine. This information is more elaborate than the information the CondorView Server receives, because the CondorView Server only receives information from the startd daemons. However, the `condor_status` command does not show the exact status of each machine, it displays `NoJob` both for machines in the state Idle and for machines in the state Owner.

The alternate CondorView Server transforms the output of the `condor_status` command to an ASCII table. This table is comparable with the output of the CondorView Server. By repeatedly running the script, for instance from a `crontab` entry, a current-status table and a machine event trace can be simulated. Both files can be explored with the CondorView Client described in the next chapter.

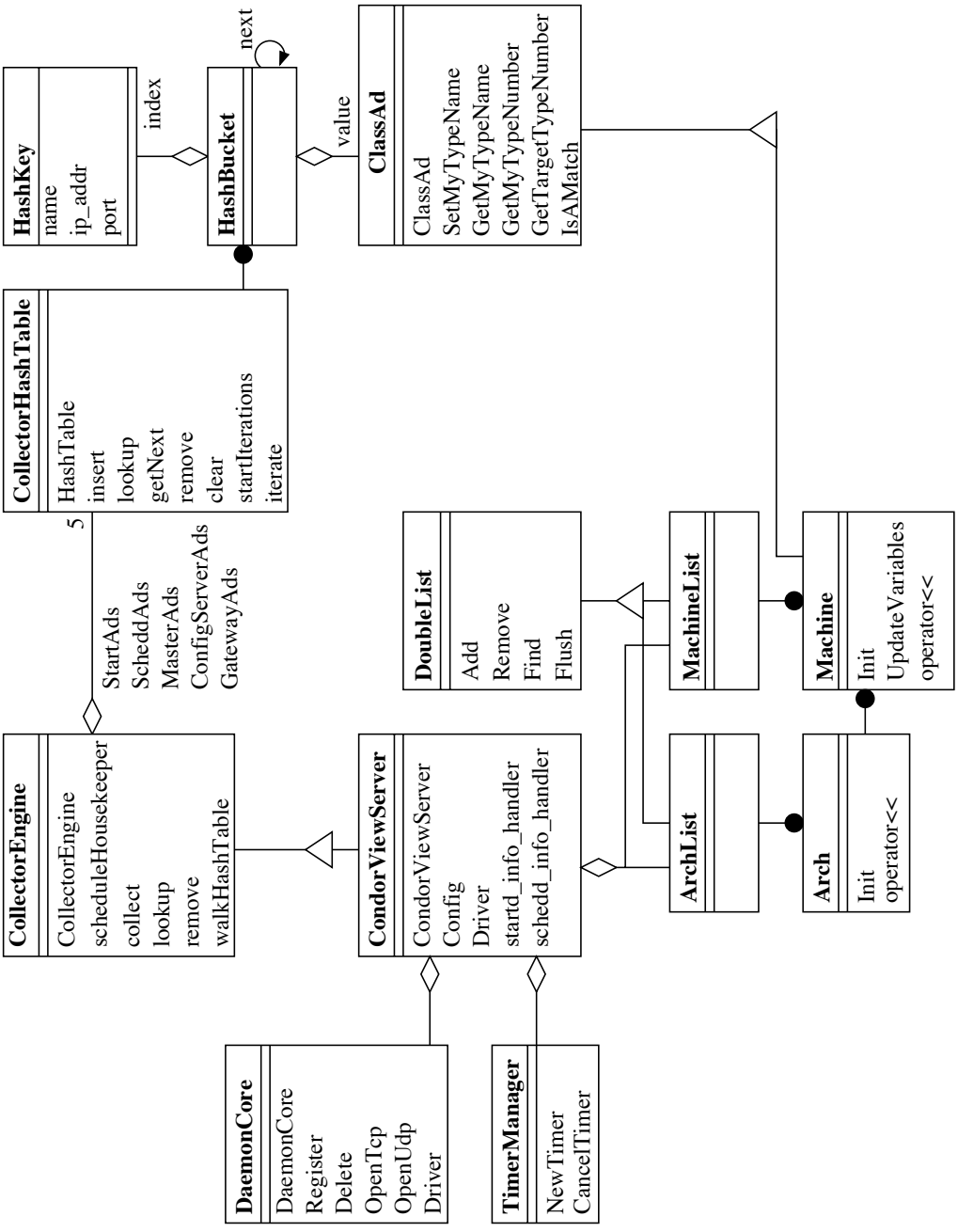


Figure 4.1: A summary of the object model of the new CondorView Server.

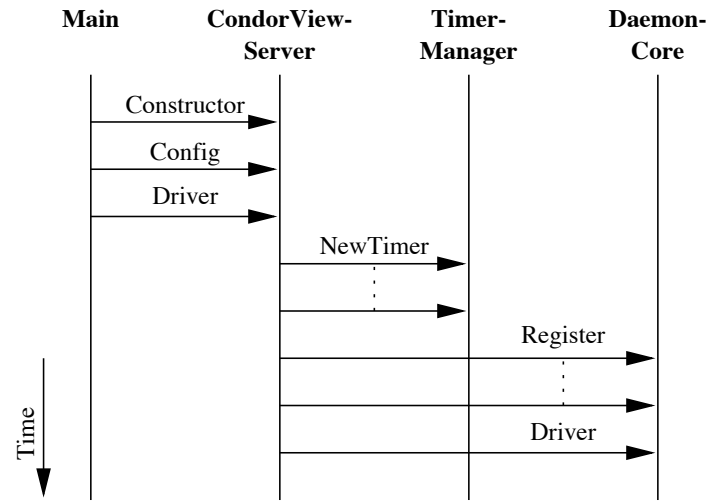


Figure 4.2: Event trace of the startup of the CondorView Server.

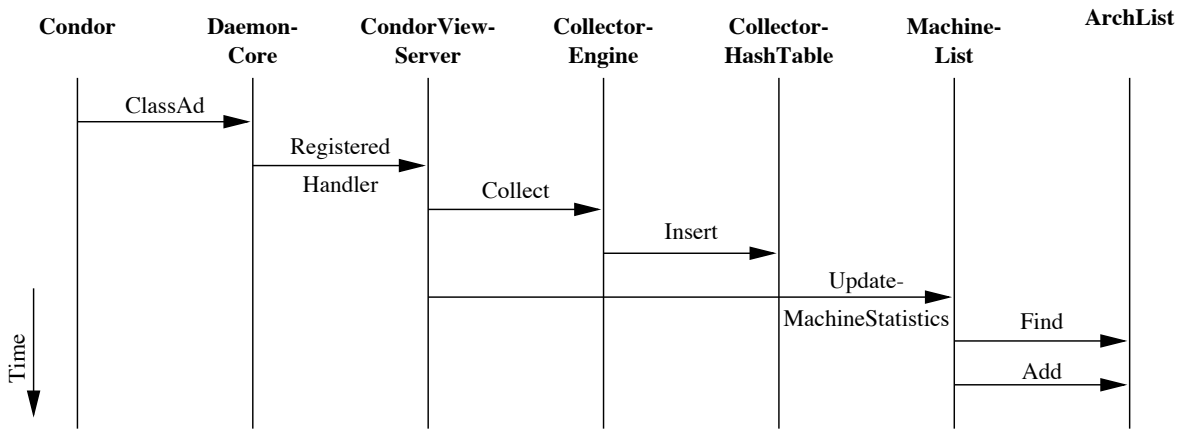


Figure 4.3: Event trace of the arrival of a ClassAd.

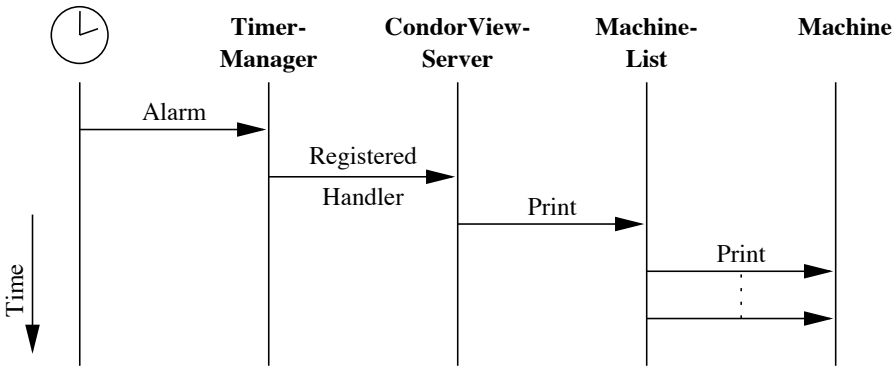


Figure 4.4: Event trace of the creations of the file for current status.

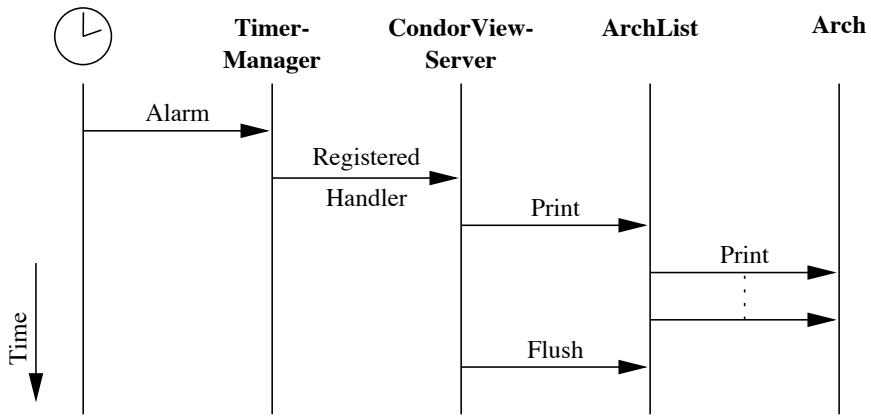


Figure 4.5: Event trace of the creation of the file for history per architecture/operating system.

Chapter 5

The CondorView Client

5.1 Functionality of the CondorView Client

When we have data collected by the CondorView Server, we can graphically explore them with the CondorView Client. In Section 2.4 we have identified the functions needed in the CondorView Client:

- Data conversion, selection, analysis, presentation and visualization.
- A graphical user interface to ease the exploration of the data.

The CondorView Client is divided into two parts. One part actually creates the visualizations by instructing DEVise to create views, windows and links. The other part is the GUI by which users can select a pool for data exploration, add and remove pools, and reconfigure a pool's settings, etc.

The first part has been divided into three view-points, which are each implemented in a separate module. Each module explores the data from a different angle, and thus different modules can be used to explore different aspects:

The current status of a Condor pool. The current status of a pool includes the state of each machine in the pool and the state of all Condor jobs currently submitted. This module can be used to explore the amount of capacity used currently, the speed of the machines used by Condor, the amount of capacity left unused, the number of Condor jobs not executing, etc.

The history of a Condor pool per architecture/operating system. The history of a pool per machine type investigates the amount of capacity Condor has made available, the type of machines (not) used, etc.

The history of a Condor pool per machine . The history of a pool per machine investigates what capacity a specific machine has added to Condor, how efficiently the available capacity of a machine has been used by Condor, etc.

In the following sections we first treat the GUI, after which we deal with the three modules, and the functions they offer.

5.2 Graphical User Interface

In this section we describe the GUI according to each window.

5.2.1 Main Window

The main window of the CondorView Client (see Figure 5.1) is similar to the main window of the previous CondorView Client. Below the Condor icon is a list of known pools. At the bottom of the window is a button bar, which contains buttons to add a new pool, reconfigure the pool, remove the pool and start the exploration of the selected pool. At the top are two buttons, one for quiting the CondorView Client, and one ('About') for checking the version of the CondorView Client and the version and build date of DEVise. A pool can be selected in the list of known pools. The selected pool can be reconfigured, removed from the CondorView Client, or explored.

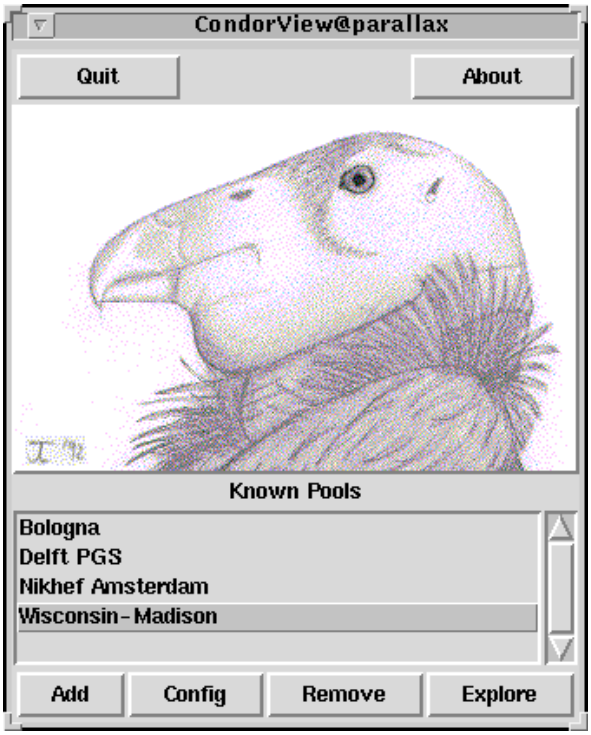


Figure 5.1: The main window of the CondorView Client.

5.2.2 Pool Configuration

When the configure button is pressed, the window in Figure 5.2 is displayed. This window contains five input fields. The first input field specifies the name of the pool, the second contains the directory where the data sets defined in the remaining three input fields are located. This directory is specified as a URL (Universe Resource Locator), and it can be on

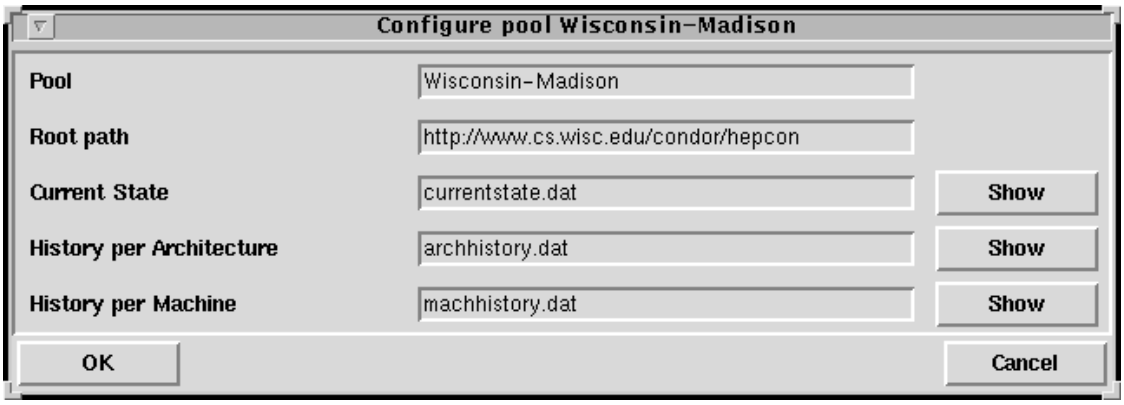


Figure 5.2: Configuration of a pool.

the internet (indicated by the string '`http:`'), or on local disk (indicated by the string '`file:`').

The remaining three input fields contain the names of the data sets for the current state, the history per architecture/operating system and the history per machine of the pool. Next to these three input fields are buttons for inspecting the contents of the entered data sets. When one of these buttons is pressed, the window in Figure 5.3 appears. When the user accepts the configuration, it is stored in the file `.condorview.config` in the user's home directory. This configuration file is read when the CondorView Client is started.

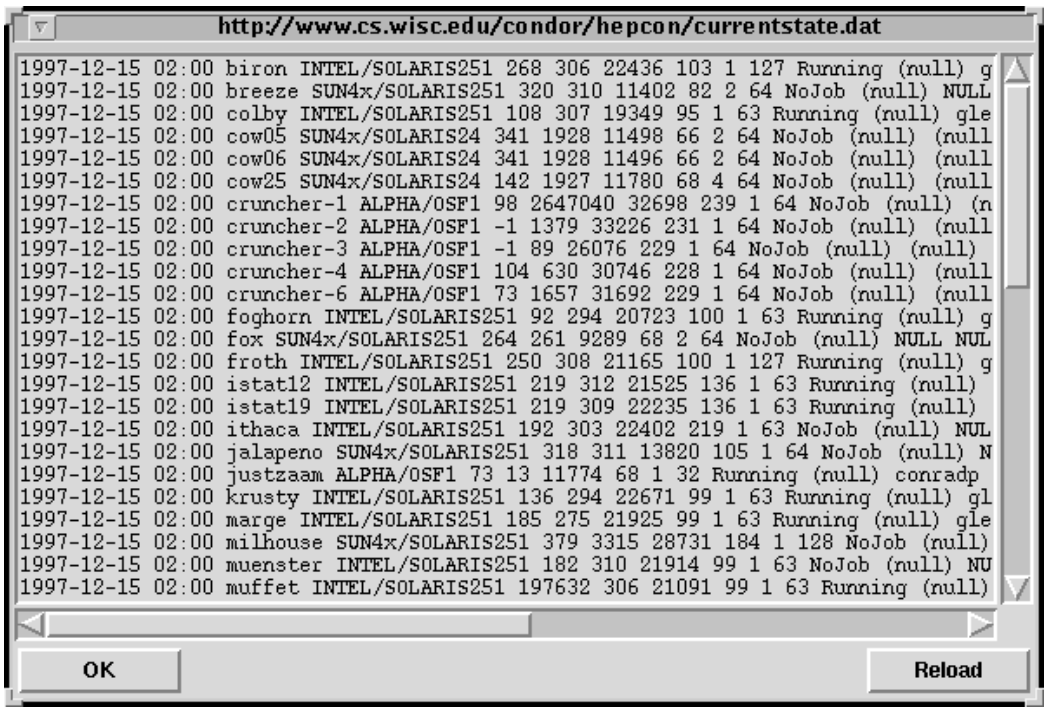


Figure 5.3: Inspection of a data set.

The 'Add' button in the main window also displays the configuration window to configure the

newly added pool.

5.2.3 Pool Exploration

Exploration of the selected pool can commence, once the 'Explore' button is pressed in the main window. The window in Figure 5.4 appears, and the user can start exploring the data sets accumulated for the selected pool. This window shows three list boxes. A module can be selected in the first list box on the left. When a module is selected, the data sets used by this module are loaded from local hard disk or over the internet. Subsequently, the CondorView Client parses the data sets to get a list of all machine names, architecture/operating system strings, etc. When the CondorView Client has finished downloading and parsing the data sets, all available functions of this module appear in the middle list box. After selecting a function, the third list box is filled with a number of variables on which the function can be executed. In the following section we treat all modules and functions.

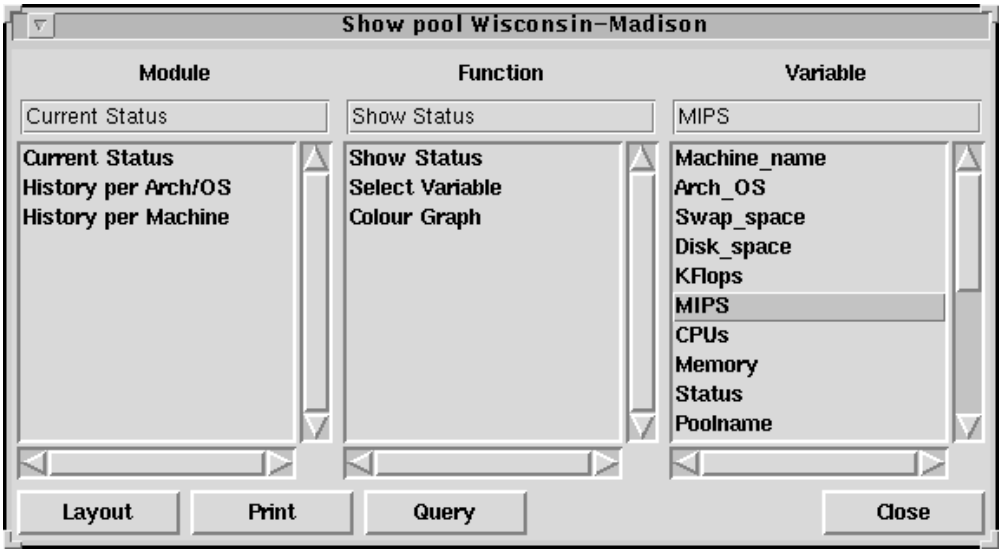


Figure 5.4: Exploration of the selected pool.

The 'Layout' button is used to signal to DEVise that the data are not to be shown, only the axis with tick marks. With the 'Print' button, all views in the window with the currently active view can be saved to file in GIF or postscript format. With the 'Query' button, the x- and y-range of the active view can be changed. In the window that shows up when this button is selected, a number of statistics of the data set depicted in the active view are also displayed. These statistics are the maximum, minimum and average value of the records displayed in the active view, and the total number of records of the displayed data set.

5.3 Exploration Modules

Each module contains a number of functions specifically made for one point of interest. Each module can visualize one or more data sets, depending on what data are needed for the point

of interest, and which data are available. New modules can easily be added to the CondorView Client. Although the functions of a module are specifically made for one module, they can easily be adapted and reused for a new module.

5.3.1 Current Status

With the module 'Current Status' the actual state of all the machines and jobs in a Condor pool can be explored.

Show Status

The 'Show Status' function plots a bar graph, where every bar represents the value of a selected variable for a machine in the pool. The bars are ordered by the state of the machines, and within each state on decreasing KFLOPS value. One or more variables can be selected, a bar graph for each variable is plotted in a window. If a graph for a variable is shown, and the variable is selected again, the graph is removed. The different graphs are linked on the x-axis; as a result, when the range of the x-axis is changed for one graph, all the others are changed to match that range. An example of this type of graph is shown in Figure 6.1.

Select Variable

With the 'Select Variable' function a range selection on a variable can be performed. The bar graphs only display event data for which the selected variable has a value within the selected range. When this function is selected a window with two views is displayed. The first view in this window contains all values of the selected variable for which an event exists, and the second view contains the selected range of the variable. These two views are connected with a cursor. The left view is the destination of the cursor, and the right view is the source of the cursor.

It is also possible to select more than one variable, as depicted in Figure 5.5. Two views appear for every variable. Each range restriction on a variable is a restriction on the events which are displayed in the bar graphs. In the selection window, the selection is narrowed down from the left to right, e.g., in the third view of the figure, only the machine names of sun4m machines are visible, and in a bar graph only the events of the five selected machines would be visible.

Colour Graph

The bar graphs pictured by the 'Show Status' are normally coloured according to the state of the machine each bar represents. With the function 'Colour Graph', the colour can be changed to match other variables than the state of the machine, for instance, the architecture/operating system of the represented machine, or the name of the remote user which currently uses the machine. The colour of only the currently selected bar graph is changed when this function is selected. In the 'Legend' window the relation between colour and the corresponding variable is pictured, for instance, when a bar graph is coloured by state, a view appears in the 'Legend' window, in which all states are listed and coloured.

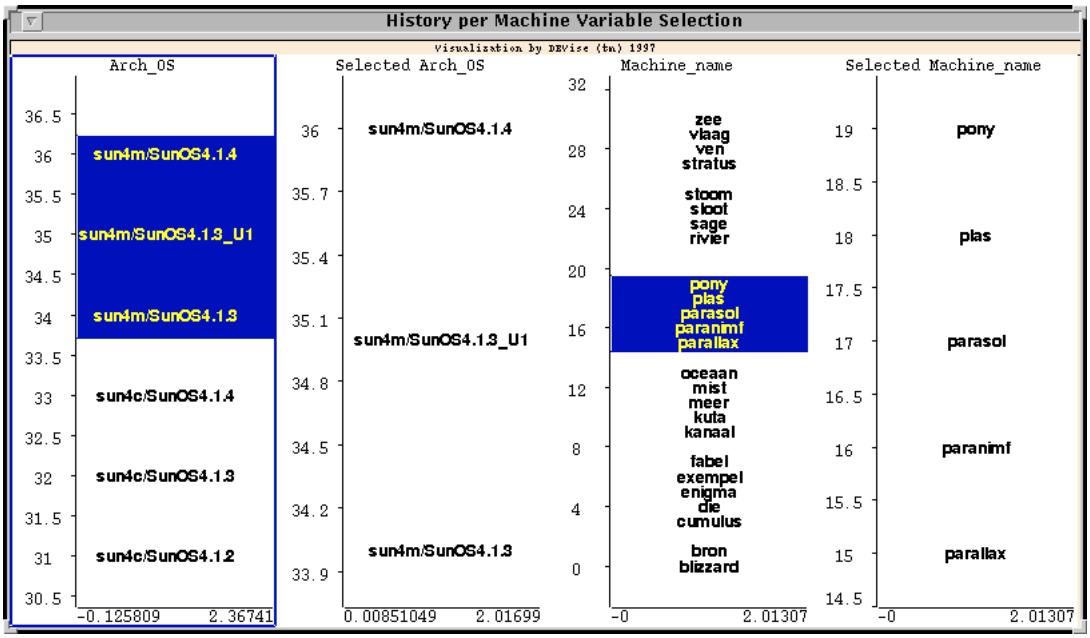


Figure 5.5: Selection of events on multiple variables.

5.3.2 History per Architecture/Operating System

The module 'History per Architecture/Operating System' is intended for the exploration of the history of a pool, based on averages per hour and per architecture/operating system of the machines in the pool. This module offers four functions, 'Show History', 'Show Overview', 'Select Variable' and 'Colour Graph'. The 'Show History' function is based on the 'Show State' function of the 'Current Status' module, but adapted for the need to show multiple events on the same x-coordinate, which is impossible with a bar graph. The 'Show Overview' function is useful in case of a large time range. Views created with this function automatically have a cursor with which the range of the views created by the 'Show History' can easily be changed. The remaining two functions, 'Select Variable' and 'Colour Graph', are identical to the functions provided with the 'Current Status' module.

Show History

The 'Show Variable' function draws a view with a scatter plot of every selected variable against time. For this function a scatter plot is used instead of a bar graph, because several architecture/operating system combinations obscure each other when a bar graph is drawn. Similar to the previous module, the different graphs are linked on the x-axis.

The 'Show History' function also allows showing two variables in one view, one from 0% upwards, and the other from 100 % downwards. This creates a clear picture when we want to compare the percentage of machines in state 'Owner' versus the percentage of machines in state 'Condor' per architecture/operating system combination. In this case, a specific architecture/operating system combination has to be chosen with the 'Select Variable' function, otherwise the bar graphs of the different architecture/operating system combinations obscure

each other. The graphs created with this function are linked with the other graphs of the 'Show Variable' function on the x-axis. Figure 6.5 pictures two examples of this type of graph.

Show Overview

With the two functions mentioned above a visualization can be constructed of data over a selected time range, but it also comes in handy to have one graph that covers a (much) larger time range. With the 'Show Overview' function, one variable can be selected for an overview. A cursor is automatically created from the overview window to the views created with the 'Show History' function. The cursor enables easy manoeuvring through the data visualized in the other views.

5.3.3 History per Machine

The module 'History per Machine' explores the history of individual machines in a Condor pool. This module offers the functions 'Show Variable', 'Show Overview', 'Select Variable' and 'Colour Graph'. These functions are described above, for the module 'History per Architecture/Operating System'.

5.4 Manipulating Visualizations made by DEVise

When a function and variable are selected in the 'Show Pool' window, the CondorView Client instructs DEVise to draw a number of windows and views. The user can manipulate the information shown within a view by using the mouse and the keyboard. In this section we discuss how these manipulations are to be performed. These manipulations are handled directly by DEVise.

Mouse buttons

A region within a view can be selected with the left and right mouse buttons. DEVise will zoom in on the selected region. When the region is selected with the left mouse button, DEVise only zooms in on an x-range, the selected y-range remains the same. When the region is selected with the right button, DEVise zooms in on both the x-range and the y-range. When the middle mouse button is pressed in a view, the x and y coordinates of the mouse pointer are shown in terms of the graph. When the mouse pointer is on top of a data point, all data of the record corresponding to the data point are displayed as well.

Keyboard

The keyboard allows a user both to zoom in and out, and to translate a view. The keys `1` and `7` are used to zoom in, the keys `3` and `9` to zoom out. By default the `1` and `7` have the same effect, that is, to zoom in on both the x and y range. However, when the key `z` is pressed within a view, the effect of these keys change for that view. The `1` is then used to zoom in on

an y-range only, and the `7` to zoom in on a x-range only. The same holds for the `3` and `9` keys for zooming out. Views can be translated with the `2`, `4`, `6` and `8` keys, and with the cursor keys.

5.5 Implementation

In this section we discuss the interface between DEVise and the CondorView Client, and the internal structure of the CondorView Client.

5.5.1 Interface between DEVise and the CondorView Client

The CondorView Client is a Tcl/Tk script [17, 22]. Tcl/Tk scripts are generally executed by an interpreter, for example `tclsh` for scripts in which only Tcl commands are used, or `wish` for scripts in which Tcl and Tk commands are used. The interpreters `tclsh` and `wish` are part of the Tcl and Tk distributions, respectively. A new interpreter can be created, to allow new commands in a Tcl/Tk script. For the interface between DEVise and the CondorView Client, we have made an adapted Tcl/Tk interpreter, which includes the command `DEVise`. Arguments to this command are sent to DEVise over a TCP connection, and the results are returned to the script. The DEVise Application Programming Interface [8] describes all possible arguments to the DEVise command. The interpreter also includes two commands to open and close a TCP connection to DEVise. The adapted interpreter with the DEVise interface is written in C, it is called `devisesh`, and its structure is pictured in Figure 5.6.

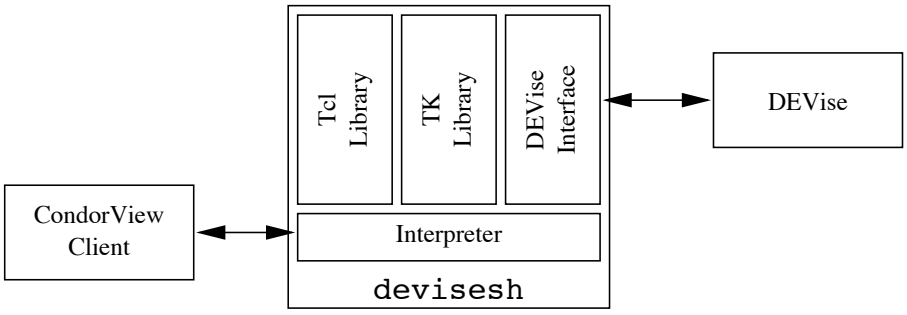


Figure 5.6: The structure of `devisesh`

Besides the command that can be sent to DEVise, DEVise also calls routines in the CondorView Client. For example, DEVise calls the routine `ProcessViewSelected` when a user selects a view, and the routine `ProcessViewFilterChange` when a user zooms in or out.

5.5.2 CondorView Client Structure

The structure of the CondorView Client is pictured in Figure 5.7. The basis is formed by two abstract data types (ADTs): an ADT to store pools and an ADT to store modules. The exploration modules use the module ADT to make themselves known to the GUI. The GUI offers the functions of the registered modules to users. When a function of an exploration

module is selected, the GUI executes the call-back routine that is registered in the module ADT. The call-back routine of the exploration module then executes a number of DEVise commands to build the views, windows, links and cursor. The exploration modules never interact directly with the GUI, and the GUI never interacts directly with DEVise. This division facilitates the development of the GUI and the exploration modules. When future developments of DEVise offer a Java interface, the GUI can be rewritten in Java, and the exploration modules can be translated (almost) automatically to Java.

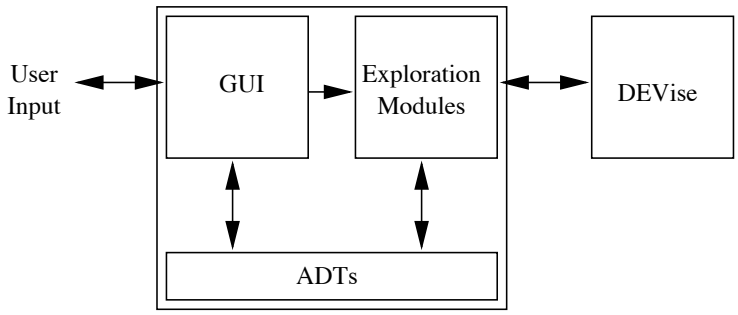


Figure 5.7: The structure of the CondorView Client.

Chapter 6

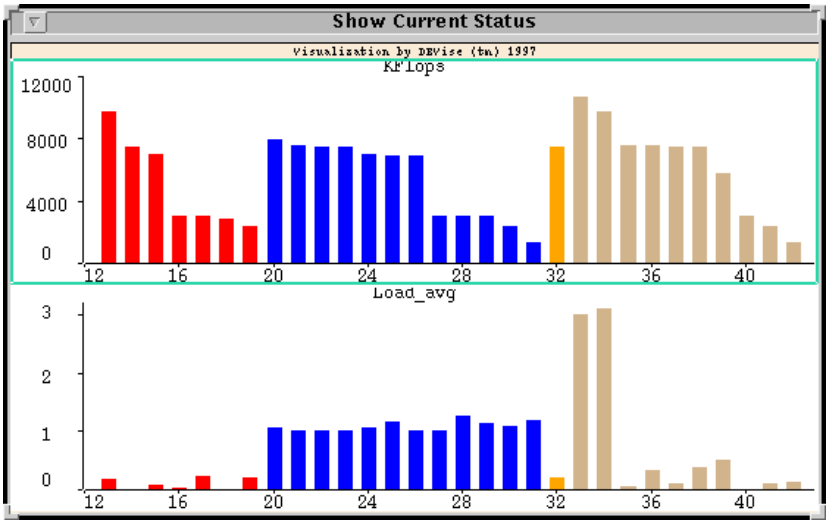
Exploration Examples

This chapter shows a number screen dumps of explorations made with the new monitoring structure. Because the explorations made by DEVise are interactive, the screen dumps in this chapter only give an idea of what is possible with the combination of the CondorView Client and DEVise.

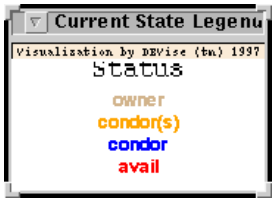
The data sets used for the explorations in this chapter are obtained at the NIKHEF pool in Amsterdam, the University of Wisconsin-Madison pool, and the INFN Bologna pool. In Amsterdam the new CondorView Server is used, in Wisconsin-Madison and Bologna the alternate CondorView Server is run every ten minutes using a `crontab` entry.

The state `condor(s)` in the legends stands for the state Suspended. The legend of the examples for the Wisconsin-Madison and the Bologna pool differ from the legend for the NIKHEF pool, because the alternate CondorView Server was used in these pools. In these pools the state `Suspended` is used instead of `condor(s)`, and the state `Nojob` indicates the states `Idle` and `Owner`.

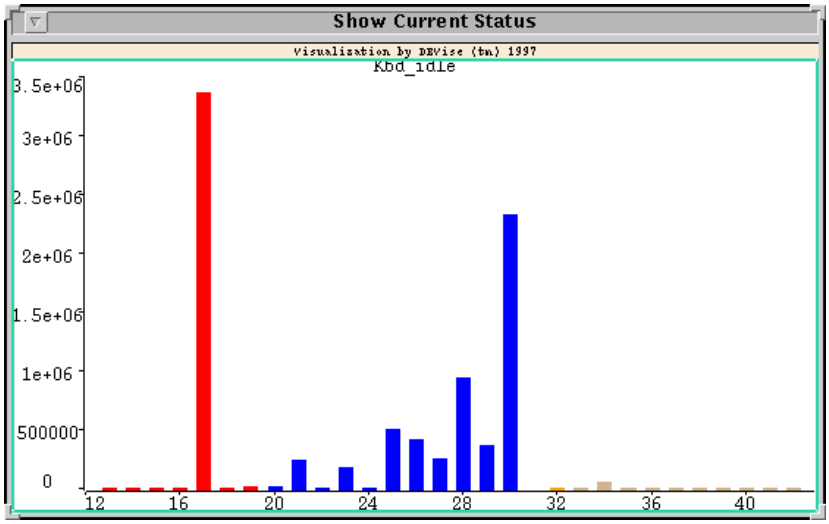
6.1 Exploration of the Current Status of a Pool



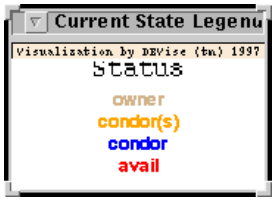
(a) The KFlops rating and the load average of all machines in the NIKHEF pool, sorted on status and KFlops rating. These views are coloured according to the status of the machines.



(b) Legend.

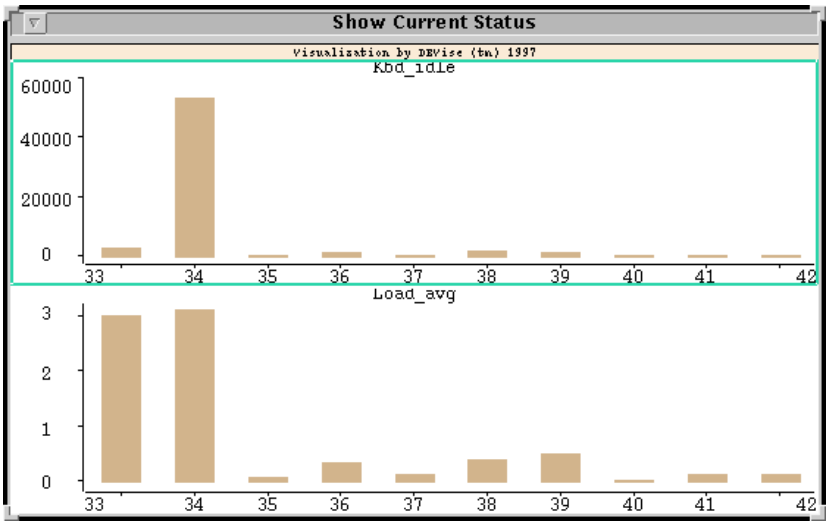


(c) The number of seconds elapsed since the last time the keyboard was touched for all machines in the NIKHEF pool. This view is coloured according to the status of the machines.

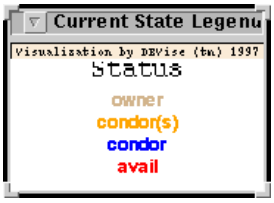


(d) Legend.

Figure 6.1: The KFlops rating, load average and keyboard idle time of all machines in the NIKHEF pool on Monday December 15, 1997, around 14:00 local time.



(a) The keyboard idle time and the load average of all machines in the NIKHEF pool in state Owner.



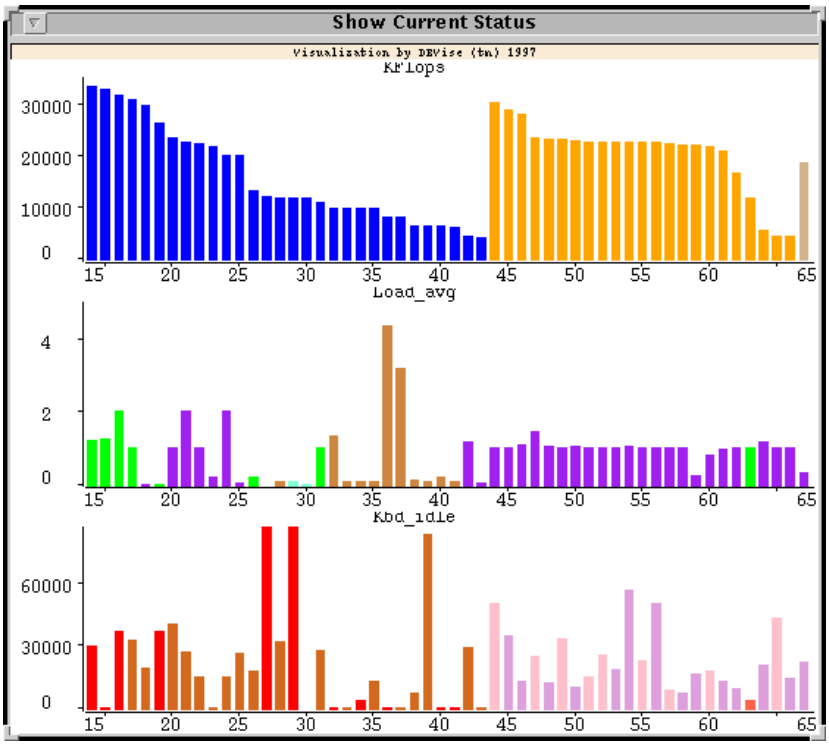
(b) Legend.

Figure 6.2: The keyboard idle time and the load average of all machine of the NIKHEF pool in the state Owner on Monday December 15, 1997, around 14:00 local time.

Figure 6.1(a) shows that approximately 40 % of the machines in the pool are being used by their owners, and that about the same number of machines are used by Condor. The figure clearly shows that the machines in state Condor are being fully used, because these machine all have a load average around 1. Only two of the machines in state Owner are used very heavily, the remaining machines probably are being used interactively. There is one machine in state Condor(s), which means Suspended.

Figure 6.1(c) shows for all machines in the NIKHEF pool the number of seconds elapsed since the last time the keyboard was touched. For most machines in state Condor, the keyboard idle time is over 48 hours (approximately 175,000 seconds). Almost all machines in state Owner have a low keyboard idle time. One machine in state available has a keyboard idle time of approximately 40 days ($\approx 3.5 \cdot 10^6$ seconds). This fact indicates that Condor can make more computing cycles available, when more Condor jobs are submitted for the architecture/operating system of this machine.

Figure 6.2(a) shows the machines in state Owner. The keyboard idle time of these machines are all short. Just one machine has a large keyboard idle time, but this machine also has a large load average, which accounts for the fact that the machine is not in state Available.

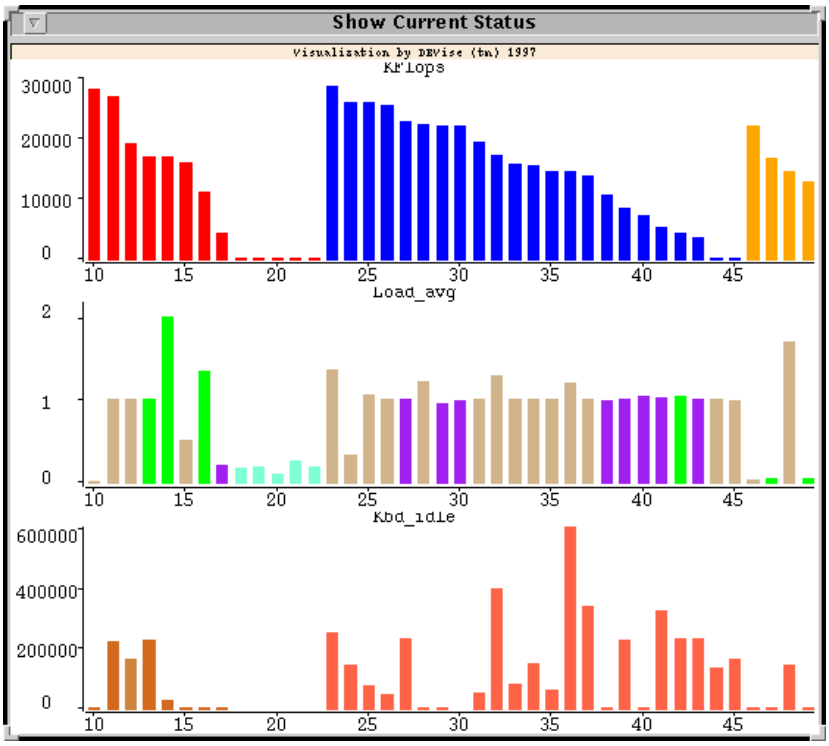


(a) The first view shows the KFlops rating of the machines in the Wisconsin-Madison pool, coloured according to the status of the machines . The second view shows the load average of the machines in the pool, coloured according to the architecture/operating system of the machines. The third view shows the keyboard idle time of the machines in the pool, coloured according to the Condor user of the machines.

(b) Legend.

Figure 6.3: Status of the machines in the Wisconsin-Madison pool on Tuesday December 16, 1997, around 04:00 local time.

Figure 6.3(a) shows in the first view that the machines in the Wisconsin-Madison pool in general are faster than the machines at NIKHEF, and that almost half of the machines is used by Condor. The second view shows that all machines except one in state Condor are Intel/Solaris 2.51 machines. Probably more computer cycles could be made available from this pool by Condor, if more jobs for ALPHA/OSF1 machines would be submitted. The SUN4x/Solaris 2.51 machines are not that interesting for Condor, because their keyboard idle times are much lower and their KFlops ratings are also lower. The scale of the third view in Figure 6.3(a) is one day (86, 400 seconds). This view clearly shows that the keyboards of most of the machines have not been touched for several hours.



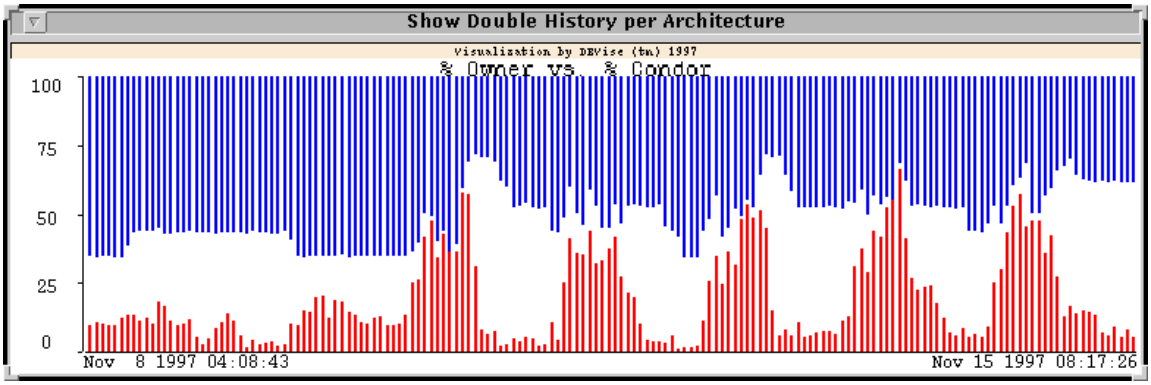
(a) The first view shows the KFlops rating of the machines in the Bologna pool, coloured according to the status of the machines. The second view shows the load average of the machines in the pool, coloured according to the architecture/operating system of the machines. The third view shows the keyboard idle time of the machines in the pool, coloured according to the Condor user of the machines.

(b) Legend

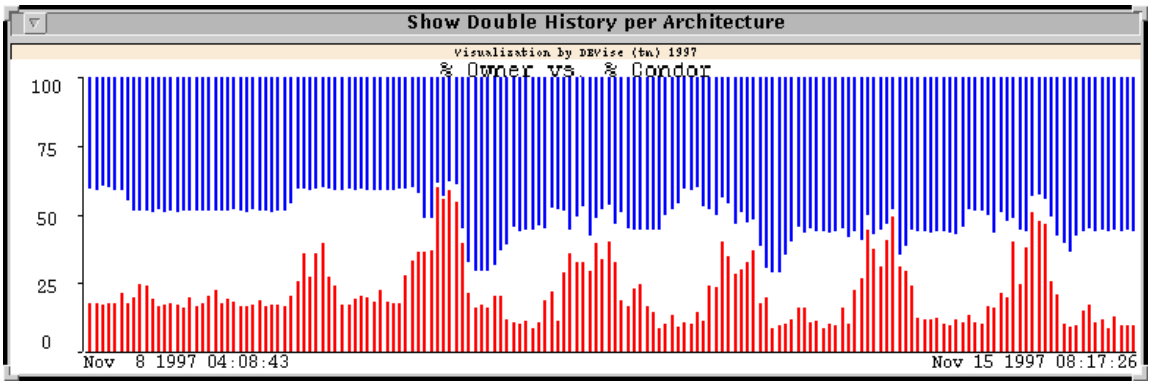
Figure 6.4: Status of the machines in the Bologna pool on Monday December 15, 1997, around 9:30 local time.

Figure 6.4 shows that the machines used most in the Bologna pool are the ALPHA/OSF1 and the Intel/Linux machines. The machines of type MIPS/Ultrix 4.4 are not used at all, and they also fail to supply the KFlops rating and the keyboard idle time. The scale of the third view is one week (604,800 seconds), so the keyboards of the machines used by Condor have not been touched for a number of days.

6.2 Exploration of the Pool History per Architecture/Operating System

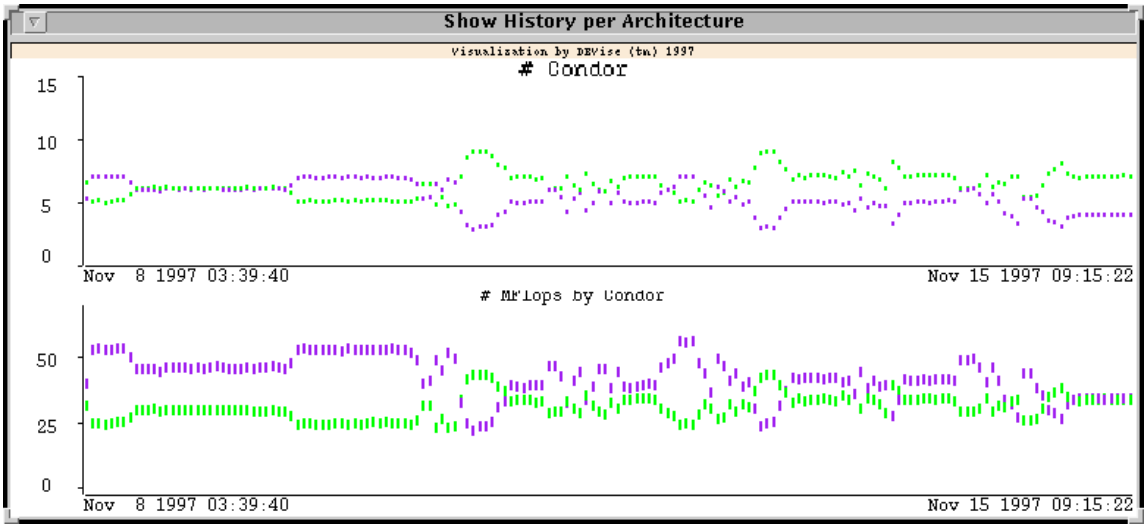


(a) The percentage of machines in state Owner (from 0% upwards) versus the percentage of machines in state Condor (from 100% downwards) over a period of one week, for the SUN4M/SunOS4.1.4 machines in the NIKHEF pool. The total number of machines of this type is 11.

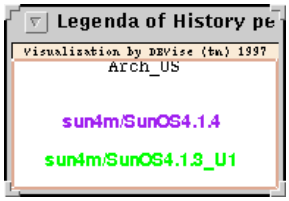


(b) The percentage of machines in state Owner (from 0% upwards) versus the percentage of machines in state Condor (from 100% downwards) over a period of one week, for the Sun4M/SunOS4.1.3_U1 machines in the NIKHEF pool. The total number of machines of this type is 13.

Figure 6.5: Percentage of machines in state Owner versus Condor for two machine types in the NIKHEF pool over a period of one week.



(a) The top view shows over a period of a week the number of machines in state Condor for the two most important architecture/operating system combinations. The bottom view shows over the same period, accumulated per architecture/operating system, the amount of computer cycles obtained, expressed in MFlops.

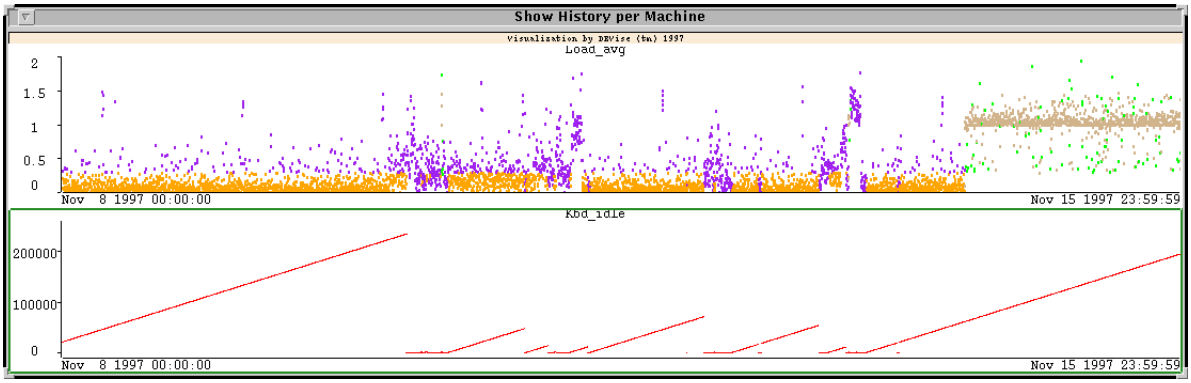


(b) Legend.

Figure 6.6: Number of machines in state Condor and the amount of computer cycles acquired by Condor for two machine types in the NIKHEF pool over a period of one week.

The Figures 6.5(a) and 6.5(b) clearly show both the day and night hours, and the week-days and week-end. Figure 6.6 shows a strange phenomenon, that is, in a number of occasions the number of machines in state Condor decreased for the SUN4m/SunOS4.1.4 machines and at the same time increased for the SUN4m/SunOS4.1.3_U1 machines, or vice versa. The number of machines in state Condor are expected to decrease and increase together when the business hours start and stop.

6.3 Exploration of the Pool History per Machine

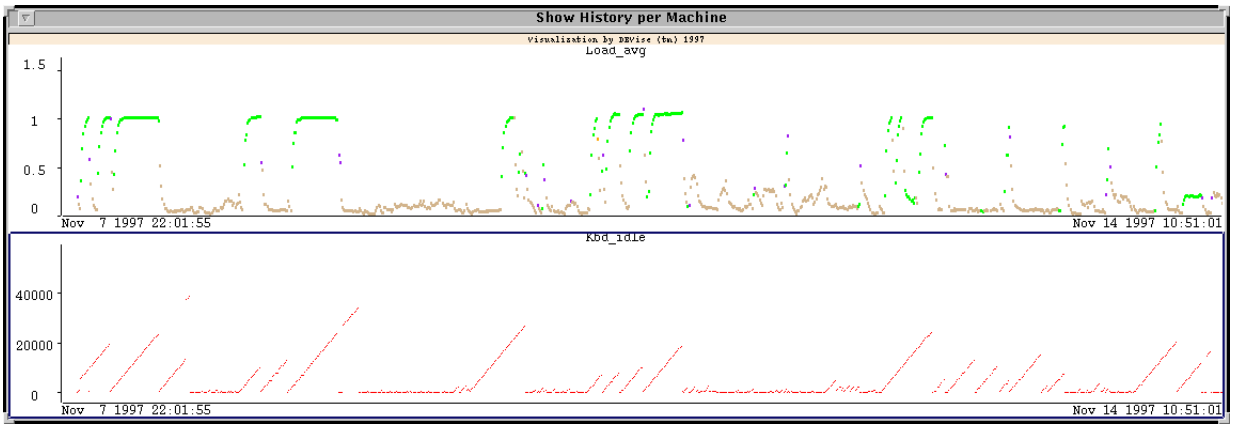


(a) The load average and the keyboard idle time of the machine `parallax` in the NIKHEF pool over a period of one week, coloured according to the state of the machine.

(b) Legend.

Figure 6.7: The load average and the keyboard idle time over a period of one week for the machine `parallax` of the NIKHEF pool.

Figure 6.7 shows that it is essential to have data on both machines and jobs. The machine `parallax` is available during most of the week, but nevertheless, Condor does not use the machine until the 14th. The history files of Condor, which are currently not available to CondorView, do not show submissions of Condor jobs on that day, so the Condor job that started running on the 14th on `parallax` was submitted at least hours earlier. So, to get a good insight into the performance of Condor, we also need a graph showing the number of Condor jobs submitted and the number of Condor jobs running.

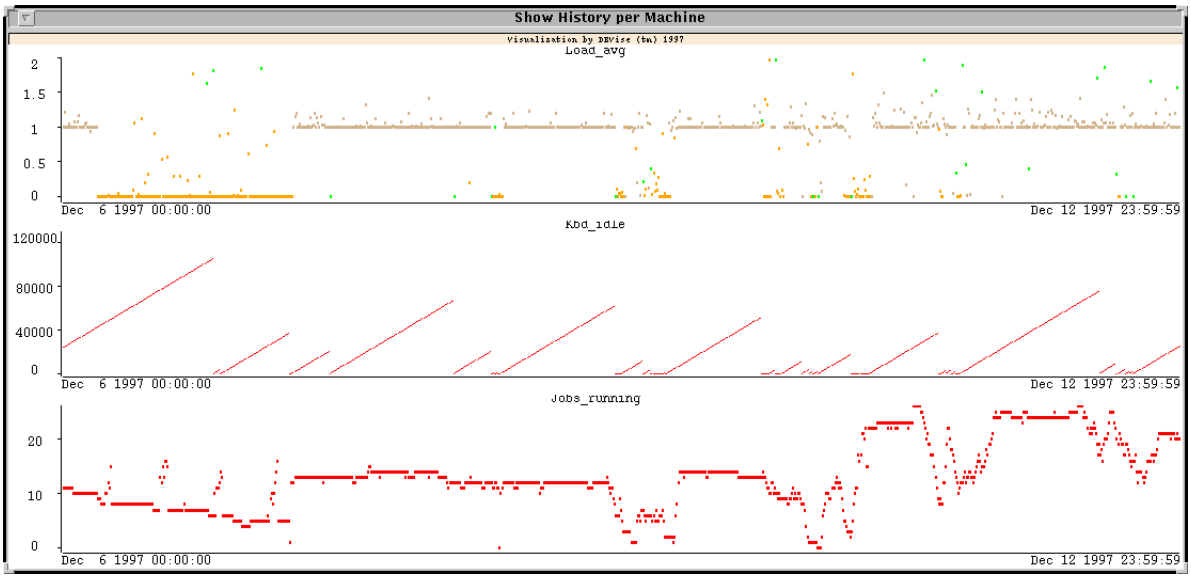


(a) The top view shows the load average coloured according to the state, the bottom view shows the keyboard idle time of the machine `fox` in the Wisconsin-Madison pool, all over a period of one week.

(b) Legend.

Figure 6.8: The load average and keyboard idle time over a period of one week for the machine `fox` of the Wisconsin-Madison pool.

A typical example of how Condor can use idle desktop machines is displayed in Figure 6.8. The machine `fox` is operated by a user on the keyboard during the business hours. In these hours, the load average of the machine is low, most of the times below 0.5. Once the user leaves the keyboard, the machine is used by Condor. Then the load average is constant at 1.



(a) The first view shows the load average coloured according to the state, the second views shows the keyboard idle time, and the third view shows the number of Condor jobs submitted on `axpbo8` and running in the Condor pool for the machine `axpbo8` of the Bologna pool, all over a period of one week.

(b) Legend.

Figure 6.9: The load average, keyboard idle time, and the number of Condor jobs submitted on `axpbo8` and running in the Condor pool for the machine `axpbo8` of the Bologna pool over a period of one week.

Most of the Condor jobs in the Bologna pool are submitted on the machine `axpbo8`, pictured in Figure 6.9. This machine is not used regularly during business hours. However, the business hours can be recognized in the last view. In four occasions the number of Condor jobs submitted on `axpbo8` and running in the Condor pool decreases, increases again during lunch hours, decreases in the afternoon, and finally increases again at the end of the business hours. All these Condor jobs use `axpbo8` for their remote system calls. According to Figure 6.9, the remote system calls of over 20 Condor jobs running at the same time do not create much load.

Chapter 7

Summary and Conclusions

A summary and the conclusions concerning the main topics of this report are presented below.

New monitoring structure

A new monitoring structure for Condor has been designed and implemented. The data collection and data visualization are no longer performed by one monolithic monitoring program. The main advantage of the new monitoring structure is the reduction of complexity by distributing different functions over different programs, and by using an external visualization program to perform generic data analysis, presentation and visualization. Another advantage is that development on the data collection and data exploration can now be performed independently.

CondorView Server

The starting point of the new CondorView Server is CondorView version 3.1, but all code for data visualization and for the complex data storage has been removed. The new CondorView Server is completely designed and implemented in an object-oriented fashion. Classes of Condor version 6 have been used, both for network communication and ClassAd storage in main memory. By using theses classes, maintenance of the CondorView Server is simplified. When new types of ClassAds are developed for the new daemons of Condor version 6, new source code for handling these ClassAds can easily be added. This is another advantage of the new CondorView Server.

Currently, only the startd daemons send ClassAds to the CondorView Server, and thus the collected data sets only contain machine information. The collected information is stored in ASCII files on disk in a number of ways: First, all received ClassAds are written to disk to form an event trace. Secondly, every ten minutes, the attributes of all machines in the pools are stored to disk to form the current status of the pool. Finally, every hour the collected data are averaged per architecture/operating system and stored to disk to form a summarized history of the pool.

Besides the CondorView Server a script is made to perform measurements on Condor by using `condor_status`. This script can be used in a pool where CondorView version 3.1 is still

running, or when the Condor account cannot be used to install the CondorView server. The script generates an ASCII output comparable to the current status file of the CondorView Server, and by running the script repeatedly the machine event trace file can be simulated. So the output of this script can be used by the new CondorView Client.

CondorView Client

The CondorView Client has been completely rewritten in Tcl/Tk. A Tcl/Tk interpreter has been made to interface between the CondorView client and DEVise. The CondorView Client offers a GUI to explore data sets collected at several pools. Before a pool can be explored, it needs to be added to the CondorView Client. After this, the names of the data sets can be configured at any time.

The visualization of the data sets is handled by three modules, for the current status of a Condor pool, for the history of a Condor pool per architecture/operating system, and for the history of individual machines. With the modules, users can explore the possibly distributed data sets of different Condor pools. Users can specify which variables of the data sets to visualize, over which time period, and graphically perform a selection on the range of one or more variables. New modules can easily be developed and added without changing the CondorView Client.

Exploration Examples

The exploration modules have been demonstrated in a number of examples. The three exploration modules each offer insight into a different aspect of the performance of a Condor pool. They also clearly demonstrate that more data should be collected by the CondorView Server, because the results of the explorations pose a number of questions that cannot be answered with the currently available data. But before the CondorView Server can collect more data, Condor needs to be adapted in order to have all daemons send their data to the CondorView Server.

Conclusions

The new monitoring structure allows more flexible monitoring and exploration of monitoring data than was previously possible. The complexity of monitoring and exploration has been split, which allows a separate development of both.

The CondorView Server is now completely designed and implemented in a object oriented fashion, and it can easily be extended for future Condor extensions, like new daemons. However, Condor needs some changes to allow the CondorView Server to receive all information. The storage of monitoring data in ASCII files instead of the Btree database developed for CondorView version 3.1 enables DEVise to read the monitoring data directly from the Internet. The Btree database system is no longer needed, because the event selection mechanism offered by this system has been made superfluous by the graphical selection capabilities of DEVise.

The CondorView Client allows the Condor users as well as the system manager to obtain a better understanding of the Condor system by offering flexible exploration modules rather than predefined graphs. The CondorView Client demonstrates DEVise to be a flexible visualization tool. Because DEVise offers a number of powerful visualization primitives, complex presentations of the monitoring data can easily be created for the CondorView Client.

Although the version of DEVise used in this master's project does not have all the features we would have liked (for instance histograms or logarithmic scales), the developments on DEVise promise these features to be part of DEVise quickly. By the separation of the CondorView Server and Client, and within the Client the separation of the GUI and the exploration modules, we can quickly profit from new developments of DEVise. For instance, when a Java interface becomes available for DEVise, the GUI of the CondorView Client can be rewritten in Java, while the exploration modules can be translated (almost) automatically to Java.

Bibliography

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995. `http://now.cs.berkeley.edu`.

[2] R.J.M. Boer. Resource Management in the Condor System. Master's thesis, Delft University of Technology, Department of Mathematics and Computer Science, The Netherlands, May 1996.

[3] A. Bricker, M.J. Litzkow, and M. Livny. Condor Technical Summary, Version 4.1b. Technical Report 1069, Computer Sciences Department, University of Wisconsin-Madison, 1992.

[4] R.L. Carter and M.E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. *Performance Evaluation*, 27&28:297–318, 1996.

[5] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation of Computer Systems*, 12:53–65, 1996.

[6] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997. To appear. `http://www.globus.org`.

[7] Andrew S. Grimshaw and Wm. A. Wulf. Legion — A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, August 1996. IEEE Computer Society Press. `http://www.cs.virginia.edu/~legion`.

[8] The DEVise Development Group. Devise Application Programming Interface. Part of the DEVise distribution, August 1997.

[9] The DEVise Development Group. Devise User Manual. Part of the DEVise distribution, August 1997.

[10] P. Homburg, M. van Steen, and A.S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland*, pages 75–82, September 1996. `http://www.cs.vu.nl/~steen/globe`.

[11] D. Koelewijn. An Artificial-Workload Generator for Condor. Research Assignment, Delft University of Technology, Department of Mathematics and Computer Science, The Netherlands, May 1997.

[12] D. Koelewijn and R. van der Put. Measurements of Network Latency and Bandwidth for Condor. *a239 course project*, Delft University of Technology, Department of Mathematics and Computer Science, The Netherlands, July 1996.

[13] M. Livny, R. Ramakrishan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated Querying and Visual Exploration of Large Datasets. In *Proceedings of the ACM SIGMOD*, May 1997. `http://www.cs.wisc.edu/~devise`.

[14] M. Livny, R. Ramakrishan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated Querying and Visual Exploration of Large Datasets (DEMO ABSTRACT). In *Proceedings of the ACM SIGMOD*, May 1997. `http://www.cs.wisc.edu/~devise`.

[15] M.W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Proceedings of Performace 1987, The 12th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 529–544, Brussels, Belgium, 1987.

[16] M.W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12:269–284, 1991.

[17] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[18] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, and J. Simon. The MOL Project: An Open, Extensible Metacomputer. In *Proceedings of the Sixth Heterogeneous Computing Workshop (HCW'97)*, pages 17–31, Geneva, 1997. `http://www.uni-paderborn.de/pc2/projects/mol`.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International Editions, 1991.

[20] M.V. van der Star. Monitoring Condor. Research Assignment, Delft University of Technology, Department of Mathematics and Computer Science, The Netherlands, September 1996.

[21] M.V. van der Star. Extended Monitoring Facilities in the Condor System. Master's thesis, Delft University of Technology, Department of Mathematics and Computer Science, The Netherlands, March 1997.

[22] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1995. Draft version available at `ftp://ftp.sunlabs.com/pub/tcl/welch/tkbook.ps.gz`.

Appendix A

CondorView Details

A.1 Installation of CondorView

The new CondorView has been tested on Sun SPARC workstations running SunOS 4.1.3_U1 and Solaris 2.5.1, in combination with Tcl/Tk version 8.0 and DEVise version 1.3.4.1.

Obtaining the Source Code

The source code of CondorView version 6.0 can be obtained from the following address: `http://pgsultra.twi.tudelft.nl/~condor/software/condorview-6.0.tar.gz`. This file contains the source code of the CondorView Server, the alternate CondorView Server, the CondorView Client, and `devisesh`.

DEVise can be obtained from the ftp-site of the University of Wisconsin-Madison in directory: `ftp://ftp.cs.wisc.edu/pub/devise/`. The version we used is located in the subdirectory `Devise-1.3.4.1/`. In this directory a number of archives are available which contain executables for several platforms, including SPARC/SunOS and SPARC/Solaris.

The source code of Tcl/Tk can be obtained from the ftp-site of the Sun laboratories, at `ftp://ftp.sunlabs.com/pub/tcl/`. The files we used are `tcl8.0.tar.gz` and `tk8.0.tar.gz`.

Compiling the CondorView Server

To compile the CondorView Server, do the following:

1. Login as user `condor`.
2. unzip and untar the source code in the file `condorview-6.0.tar.gz` with the command:

   ```
   gzip -dc condorview-6.0.tar.gz | tar -xf -
   ```

3. The source code is now located in the newly created directory `condor_view_60/`.

4. For the compilation of the CondorView Server source code, the `imake` configuration files which are also used for the compilation of Condor, are needed. These files are located in the `config/` directory of the Condor source code. Check the `Makefile` located in the `condor_view_60/` directory, whether the path to the `config/` directory has been set correctly.

5. Execute the command `make` in the `condor_view_60/` directory.

6. When the compilation finishes the CondorView Server executable `condorviewserver` is located in the directory `condor_view_server/`.

Installation and Configuration of the CondorView Server

Before we start the CondorView Server, we have to set a number of configuration parameters in the global Condor configuration file, normally located at `~condor/condor_config`. The following shows the values of the parameters, as set in the NIKHEF pool.

```
# condor_view
LOCAL_POOLNAME           = NIKHEF_Amsterdam
ALTERNATE_COLLECTOR_HOST = exempel.nikhefk.nikhef.nl
CONDOR_VIEW_SERVER_LOG   = $(LOG)/CondorViewServerLog
CONDOR_VIEW_SERVER_DEBUG =
CONDOR_VIEW_DATA         = /global/condorsrc/condorviewdata

# For all the STARTD AD's
CONDOR_VIEW_MACH_FILE    = $(CONDOR_VIEW_DATA)/machhistory.dat

# For the averages per architecture/os
CONDOR_VIEW_ARCH_PERIOD  = 3600
CONDOR_VIEW_ARCH_FILE    = $(CONDOR_VIEW_DATA)/archhistory.dat

# An current overview of the known machines by STARTD AD's
CONDOR_VIEW_CURRENT_MACH_PERIOD = 600
CONDOR_VIEW_CURRENT_MACH_FILE   = $(CONDOR_VIEW_DATA)/currentstate.dat
```

The `ALTERNATE_COLLECTOR_HOST` is the name of the machine where the CondorView Server is going to execute. This parameter needs to be set before the Condor daemons send their ClassAds to the CondorView Server. The alternate collector host cannot be identical to the collector host, because the CondorView Server cannot be hosted on the same machine as the Central Manager. In newer versions of Condor the variable `CONDOR_VIEW_HOST` should be used instead of `ALTERNATE_COLLECTOR_HOST`.

The CondorView Server can generate a lot of debugging information, when this is desired the `CONDOR_VIEW_SERVER_DEBUG` parameter should be set to `D_FULLDEBUG`. The debugging information is saved in the `CONDOR_VIEW_SERVER_LOG` file. The variable `LOG` is set to the directory where Condor saves all log files. All collected data are stored in the directory set by the parameter `CONDOR_VIEW_DATA`. In the NIKHEF pool this directory is mapped to

the URL `http://www.nikhefk.nikhef.nl/~condor/condorview/`, which makes the collected data available for public use.

The CondorView Server stores the collected data in three separate files. All Startd ClassAds are stored immediately in the file `CONDOR_VIEW_MACH_FILE`. This file is thus an event trace of machine events. After a period of `CONDOR_VIEW_ARCH_PERIOD` seconds the summaries per architecture/operating system are in stored in the file `CONDOR_VIEW_ARCH_FILE`. Every `CONDOR_VIEW_CURRENT_MACH_PERIOD` seconds a 'snapshot' of the collected machine data is saved to the file `CONDOR_VIEW_CURRENT_MACH_FILE`. This file represents the current state of the machines in the pool.

Once the Condor configuration file has been adopted, the CondorView Server can be started. This can best be done with the script `startup_server`, located in the directory `condor_view_60/condor_view_server/`. This script automatically restarts the CondorView Server when it dies.

When the CondorView Server is running, we have to restart the startd daemons on each machine in the pool, to reconfigure them. Once the startd daemons are reconfigured, they will send ClassAds to the CondorView Server.

Installation of DEVise

The installation of DEVise is straightforward and described in the user manual [9] accompanying the DEVise executable. After DEVise has been installed in a globally accessible place, a number of files have to be copied to the private home directory, if DEVise also has to be used for private visualizations. The files that have to be copied, are described in the document `doc/pubinst.txt`.

For CondorView, DEVise need only be available at a globally accessible location. Users need not install DEVise privately for the CondorView Client. At the NIKHEF pool, DEVise is installed in the directory `/global/devise/`, in the Delft Condor pool, DEVise is installed in the directory `/usr/condor/devise/`.

Compiling and Installation of TCL/TK

Tcl/Tk version 8.0 are required for the CondorView Client. To compile Tcl and Tk version 8.0, do the following:

1. Unzip and untar the source code archives with the commands:

   ```
   gzip -dc tcl8.0.tar.gz |tar -xf -
   gzip -dc tk8.0.tar.gz |tar -xf -
   ```

   The source code is now located in the newly created directories `tcl8.0/` and `tk8.0/`.

2. Enter the directory `tcl8.0/` and execute the command `configure`. When the auto-configure script has finished, enter the directory `tcl8.0/unix` and execute `make`. The Tcl library is now located at `tcl8.0/unix/libtcl80.a` or `tcl8.0/unix/libtcl8.0.a`, depending on your operating system.

3. Now do the same for the Tk source code. When the compiler has finished, the Tk library is located at `tk8.0/unix/libtk80.a tk8.0/unix/libtk8.0.a`, depending on your operating system.

The Tcl/Tk libraries are needed to compile `devisesh`, as described below.

Compiling `devisesh`

The `devisesh` sources are located in the directory `condor_view_60/devisesh/` of the CondorView source code. Before compilation, the `Makefile` has to be checked to make sure that the paths to the Tcl/Tk and X11 libraries are set correctly. When compiling on Solaris, extra libraries have to linked with `devisesh`. This can be done by uncommenting the line with the `SOL_LIBS` parameter. After this, `devisesh` can be compiled with the `make` command. Now copy `devisesh` to a directory in the path, e.g., at the NIKHEF, it is located in `/global/condor/bin/`.

Installation of the CondorView Client

The CondorView Client is located in the directory `condor_view_60/condor_view_client/` of the CondorView source code. It need not be compiled, because the CondorView Client is a Tcl/Tk script. However, it needs `devisesh` to transfer the DEVise commands to the DEVise daemon. The CondorView Client script `condor_view_60/condor_view_client/condorviewclient` can be copied to a directory in the path, e.g., `/global/condor/bin`, so that it can be started without specifying the complete path. Before we can start the CondorView Client, the environment variables `DEVISE_ROOT`, `CONDORVIEWCLIENT`, `TCL_LIBRARY` and `TK_LIBRARY` have to be set correctly. In the NIKHEF pool, the following accomplishes this:

```
setenv DEVISE_ROOT /global/devise
setenv CONDORVIEWCLIENT /global/condorsrc/condor_view_60/condor_view_client
setenv TCL_LIBRARY /global/condorsrc/tcl/tcl8.0/library
setenv TK_LIBRARY /global/condorsrc/tcl/tk8.0/library
```

By executing the script `$(CONDORVIEWCLIENT)/condorviewclient`, the CondorView Client can be started. It might be easier though to copy the script to a directory in the path, e.g., `/global/condor/bin` for the NIKHEF pool.

The configuration of the CondorView Client is read from the file `$(CONDORVIEWCLIENT)/condorview.config`. In this file, the names and URLs of the Condor pools are stored. When a user changes the configuration, the configuration file `.condorview.config` is written in the user's home directory. When the CondorView Client is started, it checks whether this file exists, and it reads this configuration file instead of the global configuration file.

A.2 Changes made in the Condor source code for the CondorView Server

A number of changes had to be applied to the CollectorEngine and the DaemonCore classes. The CollectorEngine and the DaemonCore class both used their own instantiations of the TimerManager class. The new CondorViewServer class also needs an instantiation. The TimerManager class uses the Unix signal `SIGALRM`, and when it receives this signal, it calls one of the registered handlers. Strange things happened when three instantiations of this class all use the same signal. Therefore, the CollectorEngine and the DaemonCore classes were adopted to use one TimerManager, of which they get a reference when they are created. The files `condor_collector.V6/collector_engine.C` and `condor_daemon_core/daemon_core.C` were affected by these changes.

A number of deficiencies appeared in the DaemonCore class, which needed to be solved. The changed methods `Register(Service*, int, ReqHandler)`, `Driver()` and `HandleReq(int)` are all in the file `condor_daemon_core/daemon_core.C`.

- The method `Register(Service*, int, ReqHandler)` registers message handlers in a hashed list. When subsequently a message is received, the correct handler is selected from this hashed list. However, the method fails to store a message handler when the list is empty. Therefore no message handler was stored at all.
- The method `Driver()` had two problems. The function of this method is to wait for incoming connections with the `select()` system call, and when a connection is made, to call the `void HandleReq(int)` method with the socket number of connection. Since we want to be able to accept connections on multiple sockets, the `select()` system call is used. This system calls returns the socket numbers when a connection is made, or the error code `EINTR` when it is interrupted by a Unix signal. This error indicates that nothing serious has happened, and `select()` has to be called again. This situation is handled correctly by the `Driver()` method. The first problem is, however, that occasionally the error `EINVAL` was returned by `select()`. This error indicates a negative value for one of the argument of `select()`. This is strange, because this argument is a constant. This problem has been solved by ignoring the `EINVAL` entirely, and call `select()` again when the `EINVAL` error occurs, identical to the `EINTR` error.
- The other problem is that the `Driver()` method ignored the socket numbers returned by `select()`, and assumed that the connection was made on one globally defined socket number. We adapted the method to correctly use the information returned by `select()`.
- The method `HandleReq(int)` is called by `Driver()` once a connection is set up. It then accepts the connection by calling the `accept()` system call for TCP connections. However, it failed to call `recv()`, instead of `accept()` for UDP connections. We adapted the method to use `recv()`, instead of `accept()`, thereby ignoring TCP connections entirely. because currently TCP is not used in the CondorView Server: ClassAds are sent with UDP messages.