A Highly Customizable System Monitoring and Control Tool

Sameer A. Verkhedkar
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

The management of a large collection of machines connected by a network and the software running on them is a difficult task. For example, processes may fail causing service disruptions, or users may exceed their quota of system resources causing shortages. If the site has a large number of machines, keeping track of all such problems can be very cumbersome. This thesis presents a system monitoring tool that addresses these problems. The tool monitors different aspects of the system state, such as processes running on each machine and their resource usage, and reports the information to the administrator via a graphical user interface(GUI). The tool can also be used for controlling remote machines through the GUI by starting or stopping processes, and for automatically restarting failed processes. Since the monitoring requirements differ for every system, the monitoring tool is highly customizable and extensible at a fine-grain level. This is achieved by implementing the tool as a collection of modules called micro-protocols that can be configured in various combinations to provide customized variants of the monitoring service. The implementation is based on Cactus, a framework for developing middleware offering fine-grain customizability for Quality of Service(QoS) attributes related to dependability, real-time and security in distributed systems. This thesis describes the design and implementation of an architecture for system administration based on a higly customizable system monitoring tool, and validates the Cactus approach for developing highly customizable software at the application layer.

ACKNOWLEDGMENTS

I express my heartfelt gratitude to my advisor, Dr. Richard D. Schlichting, for giving me the opportunity to work for him. It has been a wonderful experience. His guidance and encouragement have been invaluable, and most importantly, I'm eternally grateful to him for teaching me to write. I thank Dr. Matti A. Hiltunen for his insights and innovative ideas during the making of this project, and for his help during the writing of this document. I also thank Dr. Patrick T. Homer for serving on my committee.

I would also like to thank my wife, Rashmi, for her love, and patience while I was completing this work. I thank my father for making it possible for me to pursue higher education in the United States.

This work has been supported in part by Hitachi Systems Development Laboratory, by the Office of Naval Research under grant N00014-96-0207, and by the Defense Advanced Research Projects Agency under grant N66001-97-C-8518.

Contents

List of Figures

Chapter 1

Introduction

Many sites today contain networks of tens or hundreds of computers. Managing such a large collection of computers and the software on them is a challenging task generally referred to as *system administration*. Typical everyday tasks of a system administrator are to manage user accounts, ensure that all hardware and software is in working order, keep track of important background processes, and ensure an adequate supply of resources such as swap and disk space. Doing all this manually can prove to be very difficult, especially when dealing with a large collection of computers. On the other hand, system administration tasks such as these can be automated to a large extent. This thesis describes a tool that can be used for automating these tasks.

1.1 System Administration and System Monitoring

Many of the problems associated with managing a distributed system consisting of tens or hundreds of computers connected by a network can be simplified by *system monitoring*, i.e. by periodically checking different aspects of the system. When something goes wrong in a large distributed system, there can be a considerable delay to trace the problem manually and fix it. Not only that, there can also be a significant time lag before it is even realized that something is wrong.

A *distributed system monitoring* tool is capable of periodically checking the status of different aspects of the system. With such a tool, the status of all aspects being monitored can be stored in a central repository, and examined by the system administrator to get an overall picture of the status of the system. The tool can also analyze the status data and detect abberent conditions in the system and bring it to the system administrator's notice or take corrective action. For example, one common cause of a system failure is a process hanging or crashing. It is possible to detect such a condition as soon as it occurs with the help of an automated tool that monitors the status of important processes. The errant process can then be restarted manually by the system administrator or automatically by the tool. In the same way, other system parameters can be monitored such as the status of disk drives, I/O devices, and networks.

A system monitoring tool can also be used to monitor resources on each computer such as disk and swap space. For example, the tool can check disk usage per user once a day and send mail to users who exceed their disk quota. Long running processes that consume excessive swap space can be brought to the system administrator's notice or can be killed and restarted by the tool.

Another daunting system administration task is to keep track of software versions on all machines. It is possible for a system monitoring tool to check versions of software on each computer and notify the system administrator about out-of-date software. The same applies to checking and spreading configuration information, such as routing tables and printer configuration files, to the different machines.

System monitoring can also be used as the basis for increasing the *dependability* of a distributed system. Dependability is defined as the trustworthiness of a system such that reliance can justifiably be placed on the service it delivers [12]. A typical distributed system consists of many different parts, such as multiple computers, processors, device drivers, and physical devices, all of which contribute to the overall dependability of the system. Due to the large number of components, the probability of at least one component failing is high. System monitoring makes it possible to detect failures of components quickly and take corrective action.

1.2 Customizability

It is very difficult in software design to find a universal solution that is applicable in all situations for any problem, including system monitoring. One way to mitigate this problem, however, is to use *customizability* to provide choices. For example, every car comes with adjustable seats, tilt wheel, and adjustable mirrors. In this case, customizability is a necessity. Another example of customizability is a graphic equalizer, a device to tailor music to the taste of the listener. This concept is also applicable to software, and has found application in operating systems [16, 3, 6, 26], database systems [2, 24, 22] and communication services [11, 21, 14, 17, 4].

Customization is necessary but not sufficient by itself since it is not practical to build a new instance of a service every time a variation is required. The primary reason is that the number of possible variations can be so large that they cannot possibly be accommodated in one single application. For example, if you divide a group membership service into a set of properties, there are over a thousand different possible combinations of these properties [9]. The situation is even more complicated because variations in a service can sometimes have orthogonal semantics. For example, it is not possible to have an RPC service [15, 5] with both "at least once" and "at most once" semantics — one property has to be chosen over the other.

The same argument about customization applies to system monitoring. Every computer system or application differs in the resources it uses and the components that are considered critical. For example, monitoring CPU and memory usage of all running processes is useful for a cluster of computers being used as a development system with a large number of simultaneous users in order to ensure fair resource usage. On the other hand, these parameters are not important for a cluster acting solely as a webserver since there are no users competing for resources. Similarly, the status of disk drives and backup devices are important in a database system, but less so for a modem-bank server. Since it is not feasible to monitor all possible aspects of a system or take care of all conditions that may arise, it should be possible for the user of the system monitoring tool to be able to choose the parameters to be monitored. In addition, the tool should be easily extensible. In short, the monitoring tool needs to be highly customizable to be effective.

This thesis focuses on a type of customization where a customized service variant is constructed by configuring it out of smaller modules. We call such a service a *configurable service*. One common approach to achieving configurability is to use object-oriented techniques, where the application consists of a set of base classes and each class is responsible for providing certain properties to its derived classes [23]. Another approach — the Cactus approach [20] — is to implement the application or service as a collection of fine-grained modules, called *micro-protocols*, where each micro-protocol is responsible for a specific property or function of the application and interacts with other modules using an event mechanism. The micro-protocols are combined together with a standard runtime system or framework to form a *composite protocol*. This thesis uses the Cactus approach as the basis for building a configurable system monitoring tool.

Apart from catering to variations in user needs, configurable systems provide other advantages. Configurable systems are streamlined, i.e. code for features or properties not chosen in a configuration is excluded from the executable, thus reducing code size. Each property in the system also adds to the overall execution cost of the system. Thus, choosing just the properties that are necessary helps keep execution costs to a minimum. Configurable systems are by nature easily extensible. Adding new features means just adding new modules or new classes that fit in easily with the existing modules.

1.3 Thesis Contribution

This thesis addresses two issues. The first involves design and implementation of an architecture for system administration based on a highly customizable system monitoring tool. The tool consists of processes local to each node in the system called *local monitors* that gather data about different aspects of the node. Typically, local monitors track machine parameters such as process states, their CPU and memory utilization, and the status of I/O devices. A central process called the *global monitor* performs the function of analyzing the data collected by the local monitors, displaying data to the system administrator, and controlling the system by issuing commands to the local monitors.

The second issue is validation of the Cactus approach for achieving configurability in the application layer. Previously, Cactus has been used for developing middleware for fault-tolerant and real-time services. This thesis suggests that the same techniques can also be applied to develop applications.

The remainder of this thesis is organized as follows. Chapter 2 outlines the functionality of the system monitoring tool by describing the functions performed by the local and global monitors and their interaction in detail. It also discusses related work in this area and contrasts this approach with others. Chapter 3 presents the design of the tool. All the micro-protocols that comprise the global and local monitors are described in detail, together with the shared data structures and events. Chapter 4 is a guide to using the system monitoring tool. It describes the use of the CactusBuilder [10] to choose micro-protocols to customize the tool. The process of configuring the tool and a brief description of the tool's graphical user interface are also provided. Chapter 5 presents concluding remarks and possible future work in this area.

Chapter 2

Functional Overview

2.1 Introduction

The system monitoring tool consists of two major components: a centralized process called the *global monitor* and multiple distributed processes called *local monitors*. The global and local monitors collaborate to monitor a collection of networked machines. The global monitor is the primary entity of the system and the local monitors are *agents* that act on its behalf. An overview of the system is given in figure 2.1.

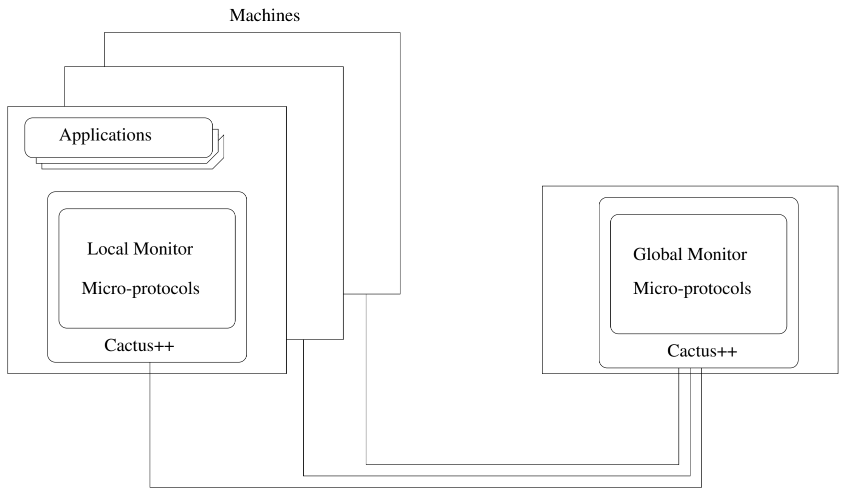

Figure 2.1: System Architecture

Each machine in the system to be monitored runs one local monitor. The local monitors track processes running on their node and gather information about them such as CPU time usage, memory usage and their current state. Other aspects of the machine such as overall process load and status of I/O devices can also be monitored. The local monitors periodically gather data and send it to the global monitor. Since the size of collected data can be large, there are provisions to apply various data filters to reduce the amount of data transferred.

The global monitor provides functionality for controlling the behavior of the local monitors through a graphical user interface (GUI). The GUI can be used for viewing data, configuring the system, and for controlling operations. The monitors use the facilities provided by the Cactus framework to send and receive messages, as well as for customizing the functionality.

2.2 Local Monitor Functions

2.2.1 Agent Functions

Local monitors act as agents on behalf of the global monitor to gather information, as well as to carry out control functions. Local monitor functions are implemented as Cactus micro-protocols, where each function is responsible for monitoring a specific aspect or parameter of the machine on which it is running. The local monitor can be customized to monitor chosen aspects depending on the micro-protocols selected to build the local monitor. The various aspects monitored by the local monitor are:

- *Process State.* The local monitor tracks the state of processes running on the node. The possible states are *running, sleeping, stopped, waiting, idle, runnable*, and *zombie*. These correspond to states maintained by the underlying OS.

- *CPU Utilization.* The local monitor tracks the CPU time utilized by each running process and reports it as a percentage.

- *Memory Utilization.* The local monitor tracks the memory utilized by each running process and reports it as a percentage of total memory.

- *Process Load.* The local monitor tracks the average process load on each node.

All this data is periodically obtained from the OS. The periodicity used for gathering data is user defined. The other functions performed by the local monitors in addition to monitoring are:

- *Starting/Stopping Processes.* The local monitor can start or kill processes on behalf of the global monitor. The global monitor provides the local monitor with the complete command line, i.e. the process name along with the arguments for starting a process.

- *Alive Signaling.* The local monitor periodically sends alive messages to the global monitor. The periodicity of these messages is user defined.

2.2.2 Data Filtering

The local monitor provides optional data filters in order to reduce the amount of data sent to the global monitor and also to give the user more control over data gathering. Each filter is implemented as a separate micro-protocol. This way, the user has a choice of selecting different data filters to create a complex filter definition. The global monitor is responsible for managing filter definitions and can control the filters remotely. If the data filters are present in the local monitor configuration, all data is passed through the filters before being dispatched to the global monitor. Currently there are two classes of data filters:

- *Numeric Filter.* These filters sift data based on numeric fields only. Currently implemented filters of this type are the CPU utilization filter and the memory utilization filter.

- *String Filter.* These filters sift data based on string fields only. Currently implemented filters of this type are the process name filter and the user name filter.

Each filter has user defined parameters that govern its behavior as follows:

- *Filter Type.* Filter type can be either *positive* or *negative*. A *positive* filter only allows data that matches the filter definition, while a *negative* filter allows only data that does not match. Filter type only applies to string filters. If not specified, a filter will be a positive filter by default.
- *Filter Logic.* Filter logic can be either *AND* or *OR*, which defines how each filter is combined with other filters. In the current implementation, the order in which the filters are specified is not significant; rather, all the *AND* filters are applied to the data before the *OR* filters. If not specified, a filter will be of *AND* logic by default.
- *Comparison Operator.* This is applicable only to the numeric filters. The operators can be =, !=, <, >, <= and >=.

The user can create a complex filter definition using these parameters. An example of combining a process name filter and a CPU utilization filter is given below; here, the process name filter is a positive filter and the CPU utilization filter is an OR filter:

Process name = "netscape","pine" OR Cpu Utilization >= 0.1

Changing the CPU utilization filter from *OR* to *AND*, and the process filter from *positive* to *negative*, the filter expression can be changed into something totally different:

Process name != "netscape","pine" AND Cpu Utilization >= 0.1

In the first case, the filter allows through any process named `netscape` or `pine` along with processes that have a CPU utilization greater than or equal to 0.1. In the second case, the filter allows through only processes not named `netscape` or `pine` that also have CPU utilization greater than or equal to 0.1.

2.2.3 Local Monitor Activation

Local monitors are activated in two steps. In the first step, the local monitors are started by the global monitor on each machine in the system. Local monitors do not start their operation immediately, but wait for a *Start Message* from the global monitor, which is the second step. Between the first and second steps, the global monitor can send configuration information such as filter definitions to the local monitors.

Once the local monitors receive the start message, actual data gathering operation is started. Each micro-protocol gathers its own data periodically, filters it, and dispatches it to the global monitor. The local monitor also responds to control messages from the global monitor.

2.3 Global Monitor Functions

The global monitor is the central entity in the system and is responsible for managing the local monitors and for interacting with the user through the GUI. The major functions of the global monitor are:

- *Local Monitor Control.* The global monitor is responsible for starting the local monitor processes on individual machines and provides controls for starting their data gathering operations.

- *Data Gathering.* The global monitor receives data messages from the local monitors, and stores the data for analysis and for display to the user.

- *Filter Definition.* The user can define the local monitor data filters through the global monitor. The global monitor sends default filter definitions to the local monitors when the local monitor processes are first started as described above. The user can also change the filter definitions dynamically during execution.

- *Change Detection.* The global monitor keeps track of changes occurring in the system by examining the data messages received from the local monitors. For example, the global monitor maintains information about new processes and about processes that exit.

- *Remote Operation of Applications.* The global monitor can send messages to a local monitor asking it to start or kill a process. There is also a process auto-restart function in the global monitor, where the user can register certain processes as restartable. When the global monitor detects the exit of such a process, it automatically asks the proper local monitor to restart the process.

Most of the functions mentioned above are accessible through the global monitor GUI. The GUI is described in detail in section 4.3.

2.4 Related Work

The concept of system monitoring is not new and many different tools exist. The one most closely related to our work is the Pulsar system [7], which consists of Tcl/Tk scripts called *Pulse Monitors*, where each pulse monitor executes Unix commands to gather information about a certain system aspect such as resource availability, program behavior, hardware behavior, and security. The pulse monitor scripts are executed periodically by a *Scheduler* that reads the execution periodicity of various scripts from a configuration file. If the monitored values exceed a threshold, the pulse monitors send updates called *alarms* to a central *presenter* that is responsible for displaying the results. The system can be easily extended by adding new pulse monitor scripts.

The System Administrator's Cockpit (*satool*), developed at the University of Colorado, Boulder, is geared towards early detection of problems occurring in groups of machines [13]. Each monitored machine runs a SNMP (Simple Network Management Protocol) agent that executes Unix scripts to gather data. A data collecting server polls the SNMP agents at set intervals and stores the data in a database. A display system written in Tcl/Tk provides a GUI for viewing data, checking data for alarm conditions, and interacting with the user. Satool is geared towards scalability and uses a hierarchical scheme for displaying host data. Extending satool involves making simple code changes to the SNMP agent, the data collecting server, and the display system.

The CARD (Cluster Administration using Relational Databases) system developed at the University of California, Berkeley, uses MiniSQL to store data and a Java applet interface to make the system accessible through the WWW [1]. Use of a relational database makes the system both flexible through new SQL queries and extensible by adding new fields or tables to the database. Data is gathered at each machine by Perl scripts. The system uses a hierarchy of databases, a hybrid push/pull data transfer protocol, and data aggregation to make the system scalable. The system also uses time-stamp protocols to detect and recover from failures.

The Rscan system, developed at Colorado State University, is capable of distributing and running scripts called *scans* on remote machines and collating the resulting reports to produce a formatted system report in HTML or plain text [19]. Each Rscan module contains multiple scans, and OS-independent and OS-dependent parts. Rscan can be configured to run particular modules on specific hosts and particular scans from specific modules. Rscan distributes scripts to remote hosts using `rsh` and `tar`, while data from the scans is returned via named pipes. Rscan has mostly been used for detecting security loopholes, but can be extended for other system administration purposes.

Commercial system monitoring packages are also available. Most of these packages are large products that concentrate on specific areas such as network management rather than system management as a whole. A few examples follow.

The SunNet Manager [25] is a commercial SNMP-based network management system available from Sun microsystems. Proxy agents at each node collect data and respond to SNMP requests. A central console displays data and can be used for configuring network devices. The consoles themselves can be connected in a hierarchical manner to scale the system to manage a large number of nodes. The system can collect data from other commercial network management systems and also allows users to customize the system by writing their own proxy agents.

HP OpenView is a product along the same lines, but has both hardware and software data collection agents [8]. The IRIX Operating System for SGI machines comes with a graphical utility called *gr_osview* that is capable of dynamically displaying host statistics such as CPU/memory usage, resource usage, and network information.

All the systems mentioned are extensible and configurable, however, unlike our tool, they use adhoc methods for achieving configurability. In contrast, Cactus++ provides a framework that is specifically designed for configurability and extensibility. Furthermore, none of the described systems provides a graphical utility for configuration. Creating a customized version involves manually changing scores of files, a process that is both cumbersome and error-prone. In contrast, our system administration tool can be easily configured using CactusBuilder, which helps the user to choose desired features and generates/modifies all files necessary to create a customized version of the tool.

Another noteworthy feature of our tool is the use of data filters and a push data transfer protocol to reduce network traffic. In a push protocol, data is sent from a source to a sink without the sink asking for it. In contrast, in a pull protocol, the sink has to request data, and, if the same data is requested periodically, as happens in a monitoring system, the request packets waste bandwidth. In our tool, the global monitor defines its data requirement through data filters and thereafter the local monitors push relevant data to the global monitor either periodically or when change is detected. Furthermore, data filters eliminate delivery of unnecessary data, while the push protocol enables data to be available at the global monitor as soon as possible. In contrast, the SNMP-based systems use a pull protocol, where required data has to be polled from data collecting agents, which implies a timelag between new data being collected and being requested and received by the sink.

The design trend in most described systems is to run data collecting scripts on remote machines and have them send the resulting data back to a central process for processing. In contrast, our tool uses an event-driven execution model, in which the local monitors store monitored data in shared data structures, and raise events that prompt several modules to process that data before it is sent to the global monitor. This design allows more processing to be done at local monitors, thus increasing system scalability by reducing the load on the global monitor.

2.5 Summary

This chapter describes the functionality of the system monitoring tool and contrasts it with other similar tools. The system monitoring tool consists of two major components: a centralized global monitor process and multiple distributed local monitor processes. Each machine to be monitored runs a local monitor process that is responsible for periodically collecting status data, such as the resource usage of running processes. Data collected by the local monitors is filtered and sent to the global monitor, which displays the data. The global monitor also provides an interface for configuring and controlling the tool.

Chapter 3

Design

The focus of the design is to make the system monitoring tool highly customizable. It should be easy for the user of the tool to include or exclude functionality, and also to extend the functionality by writing new modules that are easily combined with existing modules. This is achieved using the Cactus approach.

3.1 The Cactus Approach

In the Cactus approach to customization, an application is built as a set of modules, where each module is responsible for implementing a specific component or property of the application. These modules, called *micro-protocols*, communicate with each other through an event mechanism. Customized variants of the application can be built by configuring together different valid combinations of these prefabricated modules at compile time.

Using Cactus to implement the system monitoring tool provides a way to make it highly customizable. The user can easily create a customized variant of the tool by choosing from a set of predefined micro-protocols. The functionality of the tool can also be extended easily by adding new micro-protocols.

Two versions of Cactus are currently available. Cactus++ 1.0 is a C++ version of the Cactus framework that runs on the Solaris operating system. Cactus++ is object-oriented and is the more widely used version of Cactus. CactusMK 2.0 is a C version that runs on the MK 7.3 operating system [18] from OpenGroup RI. This version uses a system similar to the *x-kernel* [11] for constructing the network subsystem and supports real-time enhancements.

3.1.1 Key Concepts

The following are the key elements of Cactus:

- *Micro-protocol.* A micro-protocol is the basic building block. It is a collection of event handlers, where each event handler is a function that is associated with an event and is executed when the event is raised. Each micro-protocol implements a specific property of the service or application. The overall functionality of the service or application is determined by the set of micro-protocols selected.

- *Composite protocol.* A composite protocol is a runtime framework that binds a set of micro-protocols together and provides them with the common services such as an event mechanism and shared data. A composite protocol provides a *Uniform Protocol Interface* (UPI) that enables composite-protocols to be stacked.

- *Suite.* All the micro-protocols associated with a composite protocol are collected into a suite of micro-protocols. A given configuration will be a subset of this suite.

- *Events.* In the Cactus model, events are used to signify special conditions and to communicate between micro-protocols, and between micro-protocols and the runtime system. Event handlers can be bound to events, and when an event is raised, all handlers bound to that event are executed. Events can be raised in either *synchronous* or *asynchronous* mode and arguments can be passed to the event handlers. When an event is raised in synchronous mode, the invoker is blocked and all event handlers registered for that event are executed immediately. In asynchronous mode, the invoker is allowed to continue. It is also possible to specify a delay.

- *Site.* Each process, either local or remote, that has a Cactus composite protocol stack is called a site. Sites have unique identifiers that are used by the Cactus network layer for message passing.

Cactus uses a *two-level composition model*, where the first level consists of multiple composite protocols structured hierarchically. Each composite protocol can interact with the composite protocol above and below it through the composite protocol's expected interface. At the second level of composition, each composite protocol in turn contains multiple micro-protocols, where each micro-protocol is responsible for a certain property of the system.

3.1.2 Managing Configurations

Not all possible combinations of micro-protocols can be chosen for a composite protocol because of semantic constraints that determine the valid combinations. The various constraints or relations between micro-protocols are [10]:

- *Dependency.* A micro-protocol m_1 depends on another micro-protocol m_2 to function correctly, i.e. if m_1 is chosen, m_2 must also be chosen. This relationship is not associative.

- *Conflict.* A micro-protocol m_1 conflicts with another micro-protocol m_2 and one cannot function properly if the other is chosen, i.e. if m_1 is chosen, then m_2 cannot be chosen along with it and vice-versa.

- *Enclosure.* A micro-protocol m_1 implements a superset of the functionality of another micro-protocol m_2, i.e. m_1 encloses m_2 or m_2 is enclosed in m_1. Enclosures can be nested to any depth.

Relations between micro-protocols in a suite are usually depicted as a *configuration graph* [10]. An example is shown in figure 3.1. In a configuration graph, nodes represent micro-protocols and edges represent a dependency relationship, for example, micro-protocol `A` depends on micro-protocol `B`, and one of micro-protocols `C` and `D`. Multiple nodes inside an unlabeled node represents a conflict relationship, for example, micro-protocols `C` and `D` conflict with each other. A node enclosed in another node represents an enclousure relationship, for example, micro-protocol `F` is enclosed in micro-protocol `E`. A user can select a valid combination by referring to the graph or by using a graphical tool called the *CactusBuilder*. A short description of the CactusBuilder is in Section 4.1.

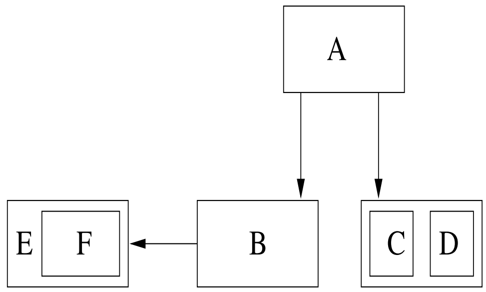

Figure 3.1: Micro-protocol Configuration Graph

3.1.3 Cactus on CORBA

A version of Cactus++ 1.0 that uses CORBA (Orbix2.2) has been used for implementing the system monitoring tool. The use of CORBA as the underlying layer provides additional useful functionality to the Cactus framework:

- *Heterogeneity.* Probably the strongest advantage of using CORBA is that heterogeneous computers can be controlled by the same system monitoring tool. Each type of computer will have a native version of Cactus and the local system monitor component running on it, where all of these are tied together by the underlying CORBA layer.

- *Remote Server Launch.* Another advantage is that it is possible to start remote servers automatically. The global monitor uses this feature to start local monitor processes on remote machines.

- *Reliable messages.* This version of Cactus uses CORBA object invocations to deliver messages between Cactus sites. This ensures reliability and message ordering at no extra cost.

3.2 Design Overview

Each portion of the functionality of the local and global monitor (chapter 2) is implemented as a separate micro-protocol. Micro-protocols within a composite protocol communicate with each other by raising Cactus events. Event handlers in the micro-protocols read and write shared data structures that are stored in the composite protocol. Explicit synchronization mechanisms are not required to arbitrate access to shared data because the Cactus model guarantees atomic execution of each event-handler function with respect to other event handlers.

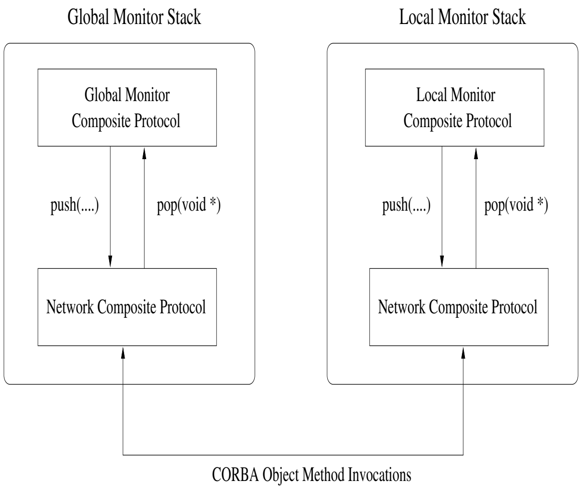

Figure 3.2: Protocol Stacks and Message Passing

The local and global monitors communicate using messages transmitted via the network composite protocol. Figure 3.2 shows the composite protocol stack at the local and global monitors. A message is sent by calling the `push()` method of network. Moreover, when a message is received by network, it calls the `pop()` method of the composite protocol above it. The network composite protocols use CORBA's remote method invocations to deliver messages.

3.3 Local Monitor

Each function of the local monitor as described in chapter 2 is implemented as a separate micro-protocol. The set of micro-protocols and their constraints are depicted in figure 3.3. The following subsections describe the micro-protocols, events, and shared data structures in detail.

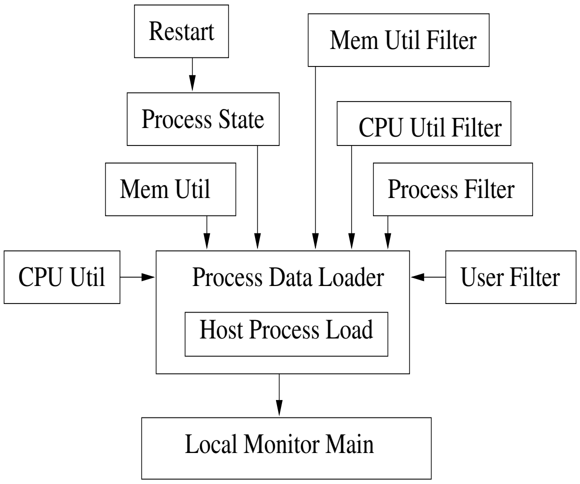

Figure 3.3: Local monitor configuration graph

3.3.1 Micro-protocols

LocalMonitorMain. This micro-protocol sends periodic `Alive` messages to the global monitor. The only parameter to the micro-protocol is an integer specifying the interval of alive signaling in milliseconds. The micro-protocol uses a local periodic event SEND_ALIVE to trigger the sending of the `Alive` messages. A periodic event is implemented by raising an event in its event handler in asynchronous mode with a fixed delay.

ProcessDataLoader. This micro-protocol periodically queries the operating system using the `top` utility to obtain information about processes running on that machine. This data is stored in the `ProcessTable` shared datastructure. This micro-protocol takes an integer parameter that specifies the query interval.

The micro-protocol uses a periodic event LOAD_PROCESS_DATA to trigger loading of new data. When data loading is completed, filtering events PROC_NORMAL_FILTER and PROC_OVERRIDE_FILTER are raised synchronously in sequence. After completion of those synchronous events, the PROCESS_DATA_LOADED event is raised to inform all micro-protocols that fresh data is available.

HostProcessLoad. This micro-protocol periodically gathers information about overall process load on the node, again using the `top` utility. It takes an integer parameter that specifies the query interval.

The micro-protocol uses a periodic event GET_HOST_PROC_LOAD to trigger the query.

ProcessState. This micro-protocol monitors changes in the state of running processes, reporting new processes and changes in state to the global monitor. It responds to the PROCESS_DATA_LOADED event raised by the `ProcessDataLoader` micro-protocol by examining data stored in the `ProcessTable` shared datastructure.

For every new process detected, it raises a GET_RESTART_INFO event to obtain command-line arguments for the new process.

CPUUtil. This micro-protocol monitors the CPU usage of processes and reports changes to the global monitor. In response to the PROCESS_DATA_LOADED event raised by the `ProcessDataLoader` micro-protocol, it examines CPU usage information about each process in the `ProcessTable` shared datastructure.

MemUtil. This micro-protocol monitors the memory usage of processes and reports changes to the global monitor. In response to the PROCESS_DATA_LOADED event raised by the `ProcessDataLoader` micro-protocol, it examines memory usage information about each process in the `ProcessTable` shared datastructure.

Restart. This micro-protocol processes restart requests from the global monitor. It responds to the RESTART_PROCESS event by starting the process mentioned in arguments to the event handler. When new processes are detected by the `ProcessState` micro-protocol, this micro-protocol queries the command-line parameters of the process and stores them in the `ProcessTable` data structure. This micro-protocol also responds to the KILL_PROCESS event, which is raised when a kill process message is received from the global monitor.

ProcessFilter. This micro-protocol implements the process name filter according to the definition provided by the global monitor. In response to the PROCESS_FILTER_DEFN event, the micro-protocol stores the new filter definition. If the filter logic is `AND`, it binds the filter event handler to the PROC_NORMAL_FILTER event, otherwise it binds the event handler to the PROC_OVERRIDE_FILTER event. When these events are raised, the filter event handler masks or unmasks process data records in the `ProcessTable` data structure.

UserFilter. This micro-protocol works similarly to the `ProcessFilter` micro-protocol, except that it filters information based on user names.

CPUUtilFilter. This micro-protocol works similarly to the `ProcessFilter` micro-protocol, except that it filters based on the CPU usage of processes.

MemUtilFilter. This micro-protocol works similarly to the `ProcessFilter` micro-protocol, except that it filters based on the memory usage of processes.

3.3.2 Events

This subsection describes the events used in the local monitor implementation. All events are raised asynchronously unless otherwise stated.

SEND_ALIVE. A periodic event indicating it is time to send an `Alive` message to the global monitor.

LOAD_PROCESS_DATA. A periodic event that indicates that it is time to load fresh data about processes running on the node.

PROCESS_DATA_LOADED. This event is raised when new data has been gathered and after the data filters have finished processsing the new data. It indicates to other micro-protocols that they can examine the data in the `ProcessTable` data structure.

GET_HOST_PROC_LOAD. A periodic event that indicates that it is time to gather host process load information from the OS.

GET_RESTART_INFO. This event is raised by the `ProcessState` micro-protocol when it detects new processes. The event handler for this event queries the OS for the information necessary to restart the new processes.

RESTART_PROCESS. This event is raised when a `Restart Process` message is received from the global monitor.

KILL_PROCESS. This event is raised when a `Kill Process` message is received from the global monitor.

PROCESS_FILTER_DEFN. This event is raised when a Process Filter definition is received from the global monitor. The USER_FILTER_DEFN, CPU_FILTER_DEFN, and MEM_FILTER_DEFN events are similar.

PROC_NORMAL_FILTER. This synchronous event is raised by the `ProcessDataLoader` micro-protocol after it has completed refreshing data in the `ProcessTable` data structure. Data filters set for `AND` logic register their handlers for this event.

PROC_OVERRIDE_FILTER. This synchronous event is raised by the `ProcessDataLoader` micro-protocol after it raises the PROC_NORMAL_FILTER event. Data filters set for `OR` logic register their handlers for this event.

3.3.3 Shared Data Structures

ProcessTable. This is a hash table of a hundred buckets for storing records containing information about processes running at the node. Each bucket in the hash table is a linked list of process records. The hash function generates the last two digits of a process identifier to produce a number between 0-99. The local monitor micro-protocols access this table to read or write process data. The declaration for this table is as follows:

```
class ProcessTable {
private:
    ProcessRec *P[PID_HASH_LEN];            // Hash table
    int old_arr, new_arr;                   // Indices for double-buffering
    int process_count;                      // Number of process records in table
    int ign_count;                          // Number of masked records
    int NumberOfProcesses;                  // Total number of processes running on node
    float LoadAvg1, LoadAvg2, LoadAvg3;     // The process load averages
    int RealMemorySize;                     // Physical memory size of node
};
```

Each process record stores relevant information about a process. Parameters that constantly change such as process state and CPU usage are stored in an array of size two. The `old_arr` and `new_arr` variables from the `ProcessTable` indicate the indices of the old and new value respectively. This is done to avoid copying of data on each refresh. The structure of each process record is as follows:

```
typedef struct process_rec {
        char ign;                        // flag used by filters to mask/unmask the record
        char name[PROCNAMESIZE];         // Process name
        char user[USERNAMESIZE];         // User name of process owner
        int pid;                         // Process Identification Number
        short state[2];                  // State of Process
        float cpu_util[2];               // CPU usage of process
        float mem_util[2];               // Memory usage of process
        unsigned int total_size;         // Size of process
        char command[COMMANDSIZE];       // Command-line arguments of process
} ProcessRec;
```

3.4 Global Monitor

Each function of the global monitor as described in chapter 2 is implemented as a separate micro-protocol. The set of micro-protocols and their constraints are depicted in figure 3.4. The following subsections describe the micro-protocols, events, and shared datastructures in detail.

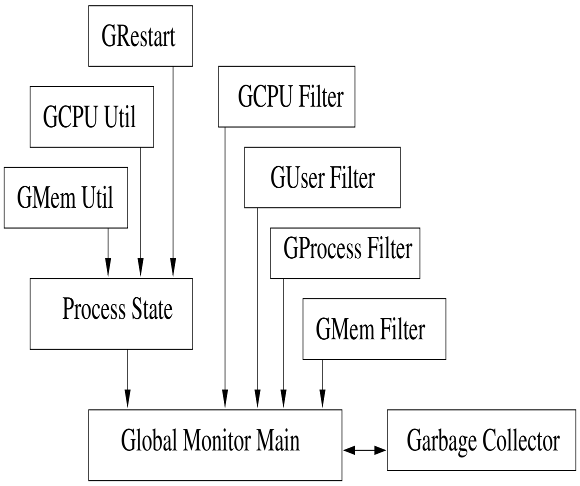

Figure 3.4: Global monitor configuration graph

The GUI code is separate from the micro-protocols and is external to the composite protocol. In this code, the GUI event processing loop is started in a separate thread. When the user performs some action in the GUI, such as pressing a button, event handlers in the GUI code are executed. These event handlers interact with the global monitor micro-protocols by raising events in the global monitor composite protocol. Since the Cactus framework only allows the composite protocol and micro-protocols to raise events, a function was added to the global monitor composite protocol to allow code outside the Cactus framework, i.e. the GUI code, to raise events.

3.4.1 Micro-protocols

GlobalMonitorMain. This micro-protocol processes the `Alive` messages that are sent periodically by the local monitors. It maintains the status of the local monitors in a shared data structure `AliveTable`. When the global monitor composite protocol receives an `Alive` message, it raises the GOT_ALIVE_MSG event, the event handler for which records receipt of the message. On a periodic event, CHECK_LIVENESS, an event handler checks if two consecutive alive messages are missing from a local monitor. If that is the case, that monitor is declared dead.

GarbageCollect. This micro-protocol frees process records for processes that no longer exist. This garbage collection is done after a delay, which is a user-defined parameter of the micro-protocol. The periodic event GARBAGE_COLLECT is used to trigger garbage collection.

GProcessState. This micro-protocol processes information received from the local monitors regarding new processes and state changes of existing processes. It stores or updates information in the shared datastructure `HostTable` and also causes updates to the GUI. The global monitor composite protocol raises the events GOT_NEW_PROCESSES, PROCESS_STATE_CHANGE, and GOT_PROC_LOAD_INFO on receipt of corresponding messages from local monitors. Event handlers for these messages store or update information in the `HostTable` datastructure. When a process ceases to exist, a RESTART_PROCESS event is raised with the process identifier as argument. This causes a check to be made whether that exited process should be restarted.

GCPUUtil. This micro-protocol processes information received from the local monitors regarding the CPU usage of processes. It stores or updates information in the shared datastructure `HostTable` and also causes updates to the GUI.

GMemUtil. This micro-protocol processes information received from the local monitors regarding the memory usage of processes. It stores or updates information in the shared datastructure `HostTable` and also causes updates to the GUI.

GRestart. This micro-protocol checks whether a process that has died is listed for restarting and sends a restart process message to the appropriate local monitor. An event handler in this micro-protocol responds to the RESTART_PROCESS event raised by the `GProcessState` micro-protocol.

GProcessFilter. This micro-protocol manages the GUI screen for configuring the Process name data filters. It also sends filter definition messages to local monitors when necessary. It responds to the event PROCESS_FILTER_DEFN raised by the GUI when the filter definition is changed by sending the definition to the proper local monitor.

GUserFilter. This micro-protocol manages the user name data filter and its GUI screen similar to the `GProcessFilter` micro-protocol. It responds to the USER_FILTER_DEFN event raised by the GUI.

GCPUFilter. This micro-protocol manages the CPU usage data filter and its GUI screen similar to the `GProcessFilter` micro-protocol. It responds to the CPU_FILTER_DEFN event raised by the GUI.

GMemFilter. This micro-protocol manages the memory usage data filter and its GUI screen similar to the `GProcessFilter` micro-protocol. It responds to the MEM_FILTER_DEFN event raised by the GUI.

3.4.2 Events

This subsection describes the events used in the global monitor implementation. All events are raised asynchronously unless otherwise stated.

GOT_ALIVE_MSG. This event is raised when an `Alive` message is received.

CHECK_LIVENESS. Periodic event to indicate that it is time to check if an `Alive` message has been received from each local monitor in the last period. If no `Alive` message is received for two consecutive periods, the monitor is declared dead.

GARBAGE_COLLECT. Periodic event that indicates time to do garbage collection in the global monitor tables.

INITIALIZE_GUI. This event is raised by the composite protocol after it has created all the micro-protocols. All event handlers for this event initialize their GUI screens.

GOT_NEW_PROCESSES. This event is raised when a `New Process` message is received from the local monitors. The GOT_CPUUTIL_INFO and GOT_MEMUTIL_INFO events are similar.

GOT_PROC_LOAD_INFO. This event is raised when a `Host Process Load` message is received.

PROCESS_STATE_CHANGE. This event is raised when information about state changes of processes is received. The event handler updates the shared table and the GUI. Terminated processes are checked for restarting and entries in the shared table corresponding to these are marked for garbage collection.

RESTART_PROCESS. This event is raised when a process has terminated.

KILL_PROCESS. This event is raised when the user selects a process to be stopped.

PROCESS_FILTER_DEFN. This event is raised when the user modifies a process filter definition. The USER_FILTER_DEFN, CPU_FILTER_DEFN, and MEM_FILTER_DEFN events are similar.

3.4.3 Shared Data Structures

HostTable. The global monitor maintains one HostTable structure for each local monitor. This table contains overall information about the host and an array of hash buckets containing process information records about each process reported by the local monitor. Records in a hash bucket are stored as a linked list. The hash function generates the last two digits of a process identifier to produce a number between 0-99. The declaration for HostTable is as follows:

```
class HostTable {
private:
        ProcessRecord *p[PID_HASH_LEN];      // Hash Buckets

        // Overall information about the host
        char hostname[HOSTNAMESIZE];
        int NumberOfProcs;                   // Number of processes running on the host
        int RealMemorySize;                  // Size of physical memory of the host
        float LoadAvg1, LoadAvg2, LoadAvg3;  // Process Load Averages
};
```

The structure of each process record is as follows:

```
class ProcessRecord {
private:
        int pid;                          // Process Identifier
        char name[PROCNAMESIZE];          // Process Name
        char user[USERNAMESIZE];          // User name of process owner
        char command[COMMANDSIZE];        // Command-line arguments of the process
        short state;                      //State of process
        float cpu_util;                   // Current CPU usage of the process
        float mem_util;                   //Current memory usage of the process
};
```

AliveTable. Array of booleans storing the status of each local monitor.

3.4.4 Summary

This chapter describes the Cactus approach in detail and presents the design of the system monitoring tool. Each micro-protocol, event, and shared data structure of the local and global monitor is described.

Chapter 4

Using the System

The system monitoring tool requires some initialization steps prior to execution. Since the monitoring tool is highly customizable, the primary task involves choosing an appropriate configuration. Other steps deal with compilation of the source code and registering the executables with the Orbix daemon.

4.1 Setting up the Monitoring Tool

The following steps are required to setup the monitoring system:

1. Choose the micro-protocols and configure the monitor instances.
2. Set constants to appropriate values in header files.
3. Make the executables and distribute them.
4. Register location with the Orbix(CORBA) daemon.

Each of these steps is now described in turn.

Micro-protocol Selection The global monitor and local monitor exist as two different micro-protocol suites, so the user needs to choose micro-protocols and configure each separately. Although this can be done by manually modifying the appropriate files, the use of the *CactusBuilder* graphical configuration tool greatly simplifies the process. To configure the global monitor, the user changes working directory to the Global Monitor source directory and runs *CactusBuilder*. On starting up, *CactusBuilder* displays a list of all micro-protocols in the global monitor suite and allows the user to choose micro-protocols and set their parameters to create a configuration. Configurations can be saved and retrieved using the *CactusBuilder* `File` menu. After creating a configuration, the user selects the `Generate C++ Files` option from the *CactusBuilder* menu to generate the files necessary for Cactus++ to build the system. An example configuration of the local monitor using CactusBuilder is shown in figure 4.1. The user selects and creates a configuration from the global monitor micro-protocol suite in a similar manner.

Set Constant Values. The user then decides the number of local monitors required and sets the constant NUM_LMON in `defs.h` to this number. The user also sets the constant NUM_SITES in `CompositeProtocol.h` to NUM_LMON+1.

Making the Executables. The user changes the working directory to the local and global monitor directories in turn and runs `make` in each directory to create the executables `gmon` and `lmon`, respectively. If necessary, the code is recompiled on different platforms to incorporate heterogeniety in the machines to be monitored. Each machine should have access to the local monitor executable `lmon`. This can be achieved by using `ftp` or `rcp` to copy the executable to remote machines, or by NFS-mounting the directory containing the executable on each computer.

Registering executables with Orbix. In Cactus++, each composite protocol stack runs as a CORBA object in separate server processes. These server processes are identified in the CORBA layer by server names that are unique strings. The monitoring system names the local monitors as `Monitor0` to `MonitorN`, where *N* is a number one less than the total number of local monitor processes. The global monitor is named `MonitorM` where *M* is equal to *N+1*. The Cactus++ system, however, uses a unique integer called `site-id` to identify each Cactus site. This integer is passed as a command-line argument to the

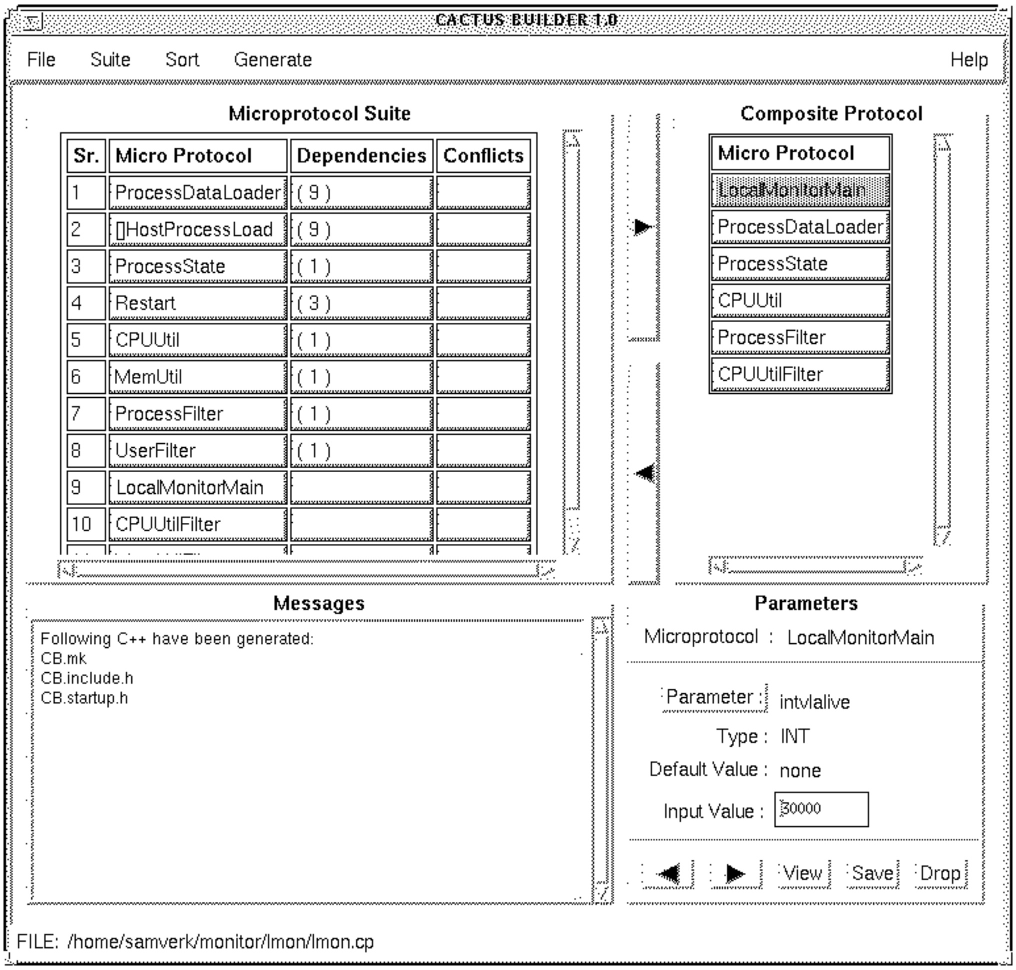

Figure 4.1: Cactus Builder

monitor executables. By registering the monitor executables with the Orbix daemon, we are essentially establishing a correspondence between CORBA server names and executables, along with the command-line arguments to be passed to the executable. The registration command also specifies the location (host) on which the executable is to be run. This registration is done using the Orbix `putit` utility, as follows:

```
putit Monitor<site> -h <hostname> <full_pathname for Monitor
Executable> <site-id>
```

This command should be executed at the machine that runs the global monitor. This registration step is required for the global monitor and each of the local monitors.

4.2 Startup and Configuration

The monitoring system is started by running the global monitor. The local monitors are then started automatically by the Orbix daemon at the machines registered in the initialization phase. As an example, the command "`gmon 2`" will start the global monitor with site identifier 2 and server name `Monitor2`. The global monitor will then start two local monitors with site identifiers 0 and 1 at the machines registered previously with the `putit` command, and with server names `Monitor0` and `Monitor1`, respectively.

As the global monitor starts, the main window of the GUI appears on the screen as shown in figure 4.2. The system is now in the *configuration phase*, during which the global monitor uploads data-filter definitions to the local monitors. At this point, the local monitors have not started their data-gathering operations, but are sending periodic `Alive` messages to the global monitor. The local monitors also respond to `Filter Definition` messages from the global monitor.

The user can now use the Edit Menu to modify the data filters, after which a start command is sent to the local monitors to start their data gathering operations.

4.3 Graphical User Interface

The global monitor provides the user with a graphical user interface to view data and control the system. The main window shows a list of local monitors and the host names of the machines on which they are executing. The screen also shows other information, such as the status of each local monitor, and the process load on each host. The main window has a menu bar on top that has menus Monitor, Edit, and View.

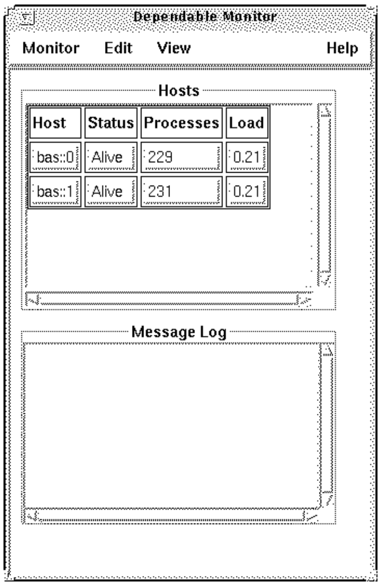

Figure 4.2: Main window

4.3.1 The *Monitor* Menu

This menu contains the global and local monitor controls. Currently, there are only three functions in this menu.

- *Start.* This is a command to the local monitors to start their data gathering operations. This command is issued in the configuration phase after all filters have been defined.
- *Save Config.* This command saves the current configuration of the global monitor in files. When the global monitor is started subsequently, it reads its configuration from these files.
- *Exit.* The global monitor sends an exit message to all local monitors and shuts itself down when this command is selected.

4.3.2 The *Edit* Menu

This menu is mainly used for defining and modifying the various data filters. It has a filter submenu that has a list of all filter types. Currently, there are four types of filters in the filter submenu, viz. process name filter, user name filter, CPU utilization filter, and memory utilization filter.

The screen for the process name filter is shown in figure 4.3. The host selector button is used to select the process filter for a particular host. The user can then add or delete names or change the filtertype and logic. To add a name, the name is typed in the textbox at the bottom, and on pressing the *Add* button, the name is appended to the list. A name can be removed from the list by selecting it from the list and then pressing the *Delete* button. The *Clear* button erases all names from the list. The apply button sends the new filter definition message to the appropriate local monitor.

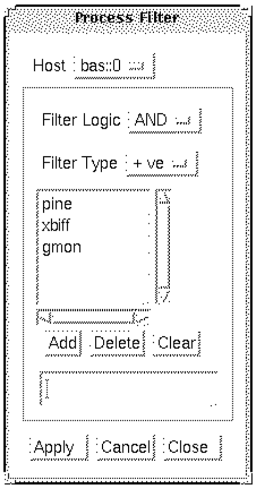

Figure 4.3: Process Filter

The screen for the CPU utilization filter is shown in figure 4.4. The usage of the CPU utilization filter is very similar to the process name filter. There is an additional selector button for the comparison function (<,>,>=,etc). The user name filter and the memory utilization filter are similar to the process name filter and the CPU utilization filter, respectively.

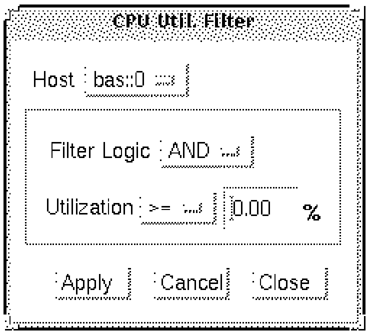

Figure 4.4: CPU Utilization Filter

The Restart List is used for registering processes for automatic restart if they crash. The screen for this is shown in figure 4.5. The usage of buttons for this screen is similar to other editing screens described before.

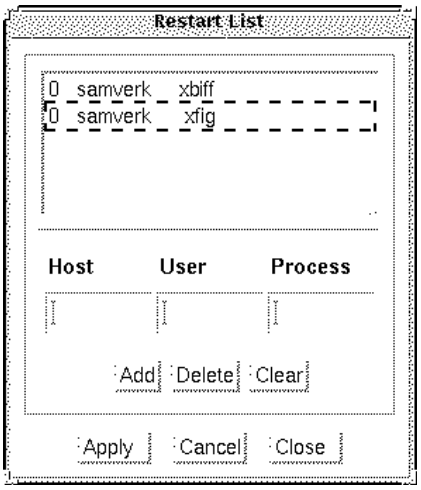

Figure 4.5: Restart List

4.3.3 The *View* Menu

The *View* menu is used for viewing data received from the local monitors. Since the system currently only deals with process information, only the screen for viewing this information is available in this menu. This screen is shown in figure 4.6. There is a host selector button for viewing processes running on a particular host. All available information about processes consistent with the data-gathering micro-protocols selected at the local monitors is shown in the screen. A process can be killed by selecting it in the list and pressing the *Kill Process* button.

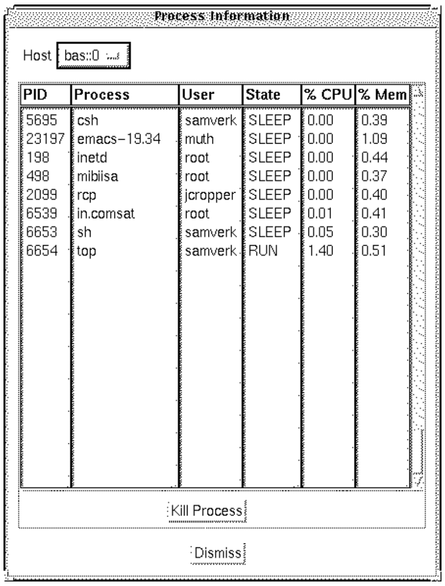

Figure 4.6: Process Information

4.4 Summary

This chapter describes the initialization and operation of the system monitoring tool. Initialization involves choosing configurations of the local and global monitors, compiling binaries, and registering the executables with the Orbix daemon. To operate the tool, the global monitor has a graphical user interface that can be used for viewing data, defining data filters, and for other control operations.

Chapter 5

Conclusion

5.1 Summary

This thesis has described a system administration tool based on a highly customizable system monitoring architecture. Although the current implementation of the tool has only a few micro-protocols that deal with a small part of system administration, the concepts developed in the thesis can easily be applied to real-life applications.

The tool has been successfully tested by having multiple local monitors running on a single machine[1]. The tool was tested with different configurations of the global and local monitors using CactusBuilder for doing the configuration.

5.2 Future Work

This thesis presents a number of opportunities for further work in following areas:

Tool Extensions. The current implementation of the tool only monitors processes and their parameters such as CPU and memory usage. The tool can be extended to other areas of system administration, including:

[1] Due to limitations of the software license, we had only one machine running Orbix.

- *Software Management.* Micro-protocols can be added that check software for proper versions and help in rapid installation/uninstallation and configuration of software.

- *Network Management.* Micro-protocols can be written to discover and graphically depict the topology of a network, monitor status of network devices, monitor network load and delays, and help in configuration of network devices. This is one of the most promising areas for applying this tool.

- *Security.* Micro-protocols can be written to periodically check machines for known security loopholes, monitor users, monitor software checksums, and check programs for improper permissions. This is also an area where the tool can be applied very effectively.

Configurable GUI. Some micro-protocols have a GUI component; for example, the `GProcessFilter` micro-protocol from the global monitor suite has a graphical interface for manipulating process filters. In the current implementation, all the GUI code is separate from the micro-protocols and is not configurable. It would be interesting to implement parts of the GUI as micro-protocols.

Failure Recovery. Although the current implementation of the tool can detect local monitor failures, it does not attempt to recover from such failures. The tool already has the mechanism to restart a failed local monitor, but attention has not been paid to other aspects of recovery such as data synchronization and consistency.

Configurability Enhancements. The version of Cactus++ used for the development of this tool allows only compile time configuration. However, work is in progress on a new version that also allows runtime configuration, i.e. allows micro-protocols to be added or removed dynamically during execution. The tool can be ported this new version of Cactus to take advantage of runtime configurability.

Bibliography

[1] E. Anderson and D. Patterson. Extensible, scalable monitoring for clusters of computers. In *Proceedings of Eleventh Systems Administration Conference (LISA '97)*, pages 9–16, San Diego, CA, Oct 1997.

[2] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, SE-14(11):1711–1729, Nov 1988.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec 1995.

[4] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[5] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[6] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, Dec 1995.

[7] R. A. Finkel. Pulsar: An extensible tool for monitoring large unix sites. *Software Practice and Experience*, 1997.

[8] Hewlett Packard. HP OpenView. http://www.hp.com/ovw/overview.htm, 1999.

[9] M. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Jul 1996.

[10] M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.

[11] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[12] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1992.

[13] T. Miller, C. Stirlen, and E. Nemeth. satool — A system administrator's cockpit, an implementation. In *Proceedings of Seventh Systems Administration Conference (LISA '93)*, pages 119–129, Monterey, CA, Nov 1993.

[14] S. Mishra. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1991.

[15] B. Nelson. *Remote Procedure Call*. PhD thesis, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[16] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[17] R. v. Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[18] F. Reynolds. The OSF real-time micro-kernel. Technical report, OSF Research Institute, 1995.

[19] N. Sammons. Multi-platform interrogation and reporting with rscan. In *Proceedings of Ninth Systems Administration Conference (LISA '95)*, Monterey, CA, Sep 1995.

[20] R. Schlichting and M. Hiltunen. The Cactus project. http://www.cs.arizona.edu/cactus/, 1999.

[21] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[22] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 85–93, Asilomar, CA, Sep 1986.

[23] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan 1991.

[24] M. Stonebraker and L. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[25] Sun Microsystems. Solstice SunNet Manager. http://www.sun.com/software/solstice/em-products/network/sunnetmgr.html, 1999.

[26] A. Veitch and N. Hutchinson. Dynamic service reconfiguration and migration in the Kea kernel. In *Proceedings of the 4rd International Conference on Configurable Distributed Systems*, pages 156–163, Annapolis, MD, May 1998.