Resource Monitoring

Abstract

Service providers must keep a watchful eye on the health of system resources to prevent inconvenient and costly service outages. Unfortunately, active monitoring of system resources can often be just as inconvenient and costly. The purpose of this paper is twofold: (1) discuss the requirements for monitoring system components, and (2) describe a statistical monitoring architecture for critical resources that is both persistent and minimizes the use of system resources.

Copyright © 2002, Intel Corporation. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0. See <http://www.opencontent.org/openpub/>.

§Names and brands may be claimed as the property of others.

Contents

Contents 3

Introduction 3

- **Flawed Approaches to Monitoring** 3
  - Non-production Components 3
  - Too Many System Resources Required 3
  - Source-Level Integration 3
- **Requirements** 3

Analysis 3

- **System Resources** 3
- **Automated Agents** 3
- **Statistics and Monitors** 3
- **Summary** 3

Options 3

Implementation 3

- **Resource Monitor Architecture** 3
- **Resource Monitor Daemon Design** 3
  - Daemon Object Model 3
  - Class Hierarchy 3
- **Application Interfaces** 3
  - Browsing 3
  - Statistic 3
  - Monitor 3
  - Monitor Configuration and Control Structures 3
- **Subsystem Interfaces** 3
- **Process Flows** 3
- **Consumer Applications** 3
- **TODOs** 3

Conclusion 3

Appendix A: References 3

Appendix B: Abbreviations, Acronyms and Definitions 3

Introduction

Systems hosting critical subscriber services cannot fail. The impact of failed systems is lost revenue, reduced subscriber satisfaction and subscriber turnover. All of these outcomes are costly for the service provider.

To reduce the potential for system failure, carriers and service providers use automated software and hardware agents that sense resource conditions, generate event indications, review system and network management policies, and select avoidance or recovery actions that will keep their services running at the highest availability level possible. Hardware, driver, kernel and application resources are all within the accessible scope of the operating system. Software sensors, or monitors, are often developed to sense changes in status or to report values of statistics kept by the resource. These sensors form the foundation of effective resource monitoring.

This paper will explore approaches to resource monitoring, compare various resource monitoring implementations, and present the architectural details of the Resource Monitor implementation.

Flawed Approaches to Monitoring

Software components that monitor system resources are not novel. Many software systems contain components that monitor system resources, but many of these implementations are flawed in some way.

Non-production Components

Since monitoring may cause sub-optimal system performance, some implementations require that a separately compiled version of the resource software be used when monitoring is needed. This implies that the monitoring-enabled code is probably not present under "production", or normal, use. Unfortunately, service providers are keenly interested in abnormal conditions that occur during production use of the system. If the monitoring is not active during production use, then event indications cannot be generated, nor can data be captured for offline analysis.

Too Many System Resources Required

In some cases, monitoring software components are not wise users of system resources. Rather than develop a design that is cognizant of system resource use, many developers take the "non-production" component approach so that the production system performance is not affected. In other cases, the monitoring may be embedded in the resource software implementation and does not provide any tuning capabilities.

Source-Level Integration

In many cases, the monitoring software component was developed for a particular type of system resource—for example, disk, network, or memory—or was developed as part of a single, specific network management framework, such as the Simple Network Management Protocol (SNMP) or Common Diagnostics Model (CIM). While the implementation may work well for the single resource type, it poses a problem for (1) system integrators that want to monitor *all* of the system resources from one management application, or (2) multiple applications that need access to a single resource type. Also, if a new algorithm for monitoring is required, the service provider or integrator must approach a third software development party to discuss adding the new functionality. This change not only increases time-to-market but needlessly introduces possible faults into the system. This same access to the source code would be needed to add new resource types to be monitored.

Requirements

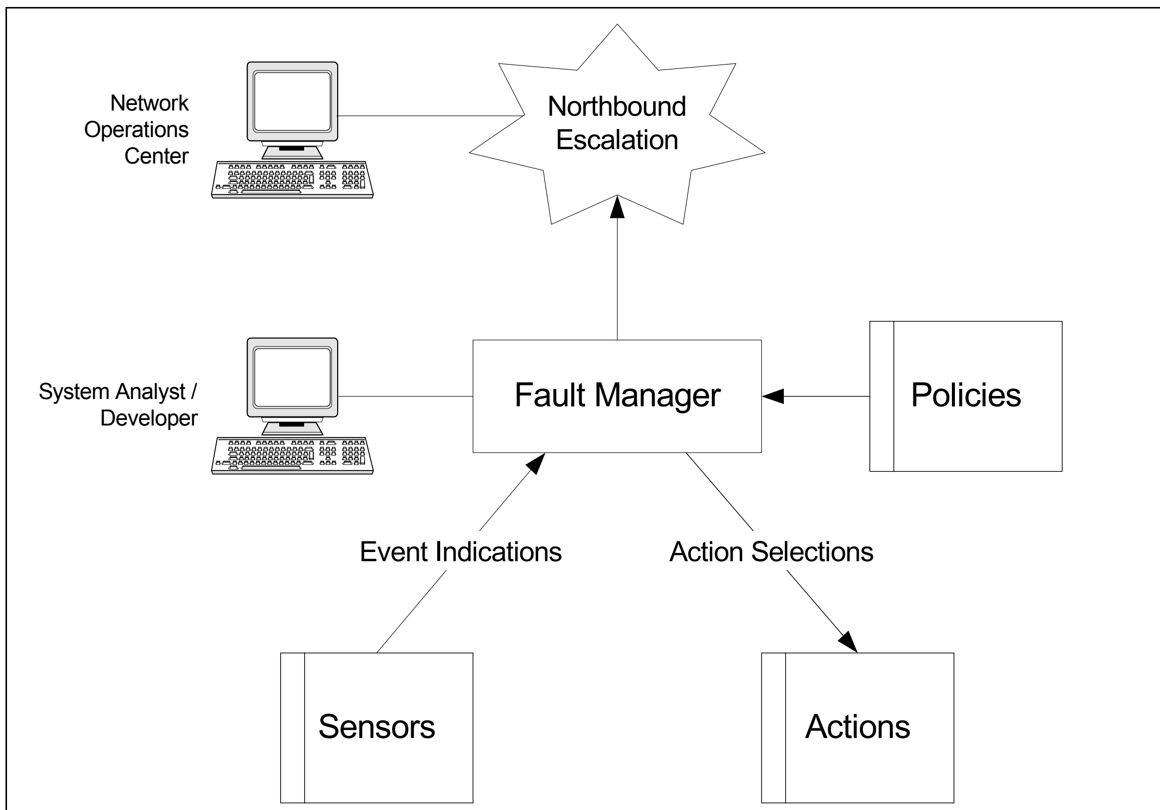
Given these potential problems, a resource monitoring software architecture is needed that:

- Monitors resources using production components.
- Uses system resources wisely.
- Allows runtime extension of monitoring to include new resource types at the binary component level, both at startup and during the life of the monitoring process.

Analysis

Resource monitoring makes up half of the foundation of the classical fault management model (see Figure 1). Sensors, or monitors, are data collectors that generate event indications, or events, for use by fault managers. Fault managers may be in the form of user applications that present the event data and allow the user to take action manually, or may be implemented as automated agents that take selected actions based on pre-configured policies related to the event data.

Figure 1 Classical fault management architecture



System Resources

Critical system resources can be found at all levels of the system but in each case, a software component must implement the monitoring function so that event indications can be generated. System resources can be broken down into the following categories:

- Hardware Resources – Typically, a driver will provide the programming interface needed to access statistics managed within hardware. In some cases, software components within the operating system track usage, as in the case of memory usage or CPU usage. Examples of hardware that may track statistics internally are:
    - Network adapters
    - System baseboards
    - Central processing units (CPUs)
- Operating System Resources – The operating system manages logical partitions of hardware resources. These “logical” resources provide insight into potential operating system problems. Examples of operating system resources are:

    - Buffers
    - File systems
    - Memory
    - Processes
- Application Resources – Application services may also maintain counters that indicate the health of the service. Applications are sometimes instrumented with libraries for retrieving this information or may keep status information in files that are updated regularly. Examples of application services are:
    - Protocol gateways, such as H.323 gateways and gatekeepers, Session Initiation Protocol (SIP) proxy servers
    - Media applications, such as Internet servers

Since monitors are implemented in many levels of a system, it becomes difficult to write maintainable applications that perform fault management. To facilitate the development of maintainable fault management applications, an architecture is needed that provides (1) an application interface that abstracts the monitoring functionality, (2) an out-of-process service that performs efficient monitoring on behalf of one or more applications simultaneously, and (3) a programming interface that allows the addition of new resource types to be monitored.

Automated Agents

Automated agents are typically components of a network management solution and are often based on standard protocols for management, such as the Distributed Management Interface (DMI), CIM, or SNMP. Each of these standards defines conventions for describing manageable entities, attributes and controls presented by those entities, and event indications generated by those entities. To facilitate the development of fault management agents that conform to one of these protocols, an architecture for monitoring is needed that models the monitoring concepts in a fashion that is easily adaptable by these protocols.

Statistics and Monitors

A component will often have the capability to generate unsolicited event data whenever a condition exists that needs attention. However, often the component has a measurable aspect that, if monitored, can indicate the existence or probability of a problem, but no proactive mechanism is available to initiate the event. In these cases, a software facility is needed that can monitor the value of a statistic and generate events when the value of that statistic meets a specified condition.

Statistics are measures associated with a system component. Statistic types range from simple counters to more complex, “second-order” statistics.

Counters are integer values that are incremented or decremented in one direction. An odometer in a car is an example of a counter—as the car is driven, the odometer increments the odometer once for every mile (or kilometer) the car moves. In networking protocols, counters are often used to count number of packets transmitted or number of packets received with errors.

Gauge statistics are numeric values that are incremented and decremented so that values move up and down over time. Temperature, speed, and voltage are all measured with gauge statistics.

Second-order statistics are those values that are derived from measurements from more than one sensor, from data collected over time, or from components where it is impossible or infeasible for the component to provide the statistic as a sampled value.

Monitors are components that observe the value of a statistic on periodic intervals and test the value of the statistic to see if it has reached a value that is of interest. If so, a monitor is configured to take an action. A thermostat is an example of a monitor—when the temperature of a room reaches a certain temperature, the thermostat is programmed to switch on an air conditioner or a heater.

Monitoring algorithms vary. A threshold monitor will test the value of a statistic and generate an event notification whenever the statistic value crosses some preset threshold value. Given the nature of this algorithm, it is most applicable to gauge statistics—those whose value moves up and down. Likewise, the watermark monitor type is

appropriate for gauge statistics since it records the highest or lowest value of a statistic within its active lifetime. For counters, monitoring algorithms that test conditions based on a rate of change, like the "leaky bucket" algorithm, are applicable. A leaky bucket monitor compares the delta in a counter value to an expected delta, over a sliding window of counter increments, and generates an event when the actual delta exceeds the expected delta.

A monitoring architecture should be based on two basic object classes, statistics and monitors. A monitor's algorithm should be available to associate with any applicable statistic. For example, a user should be allowed to activate a threshold monitor for any gauge statistic available within the scope of the monitoring facility.

The differentiation of statistics and monitors is important when designing reusable software components for a resource monitoring system. As the concepts of monitoring become more complex, it is crucial to model them accurately for maximum reuse.

For example, consider the "ratio" concept. A ratio is a comparison of two values. In some monitoring cases, the ratio may be the value of a statistic over a constant value. In other cases, the statistic of interest may be the ratio of two currently monitored statistics. If the implementer chose to model a new type of monitor as the ratio of two specific statistics, then the new monitor would have limited application. On the other hand, if this new "ratio statistic" was modeled as a second-order gauge statistic, then existing monitor implementations would be immediately usable.

Summary

In summary, the requirements of a robust resource monitoring facility are:

- To present an interface that abstracts statistical information and monitors implementations.
- To present an interface that extends capabilities to new resource types at runtime.
- To present event mechanisms via standard event management components.
- To present an object model that maps easily into network management frameworks.

Options

Many Linux§ system management components address aspects of resource monitoring and, in particular, statistics monitoring. While all of these components are valuable elements of a complete system management suite, none achieve all of the key requirements for a resource monitoring service as presented in the previous summary. The following is a review of current monitoring facilities, and presents information about the features of each, as well as the location of the open-source project.

Resource Monitor

- The Resource Monitor is an open-source resource monitoring implementation. The "Implementation" section of this paper describes the Resource Monitor in detail. The following bullets summarize the functionality of the Resource Monitor.
- A daemon service that tracks resource statistics and monitors on behalf of applications
- User APIs: shell/XML utility; C++/C libraries
- User choice of daemon-resident or kernel-resident monitoring (kernel-resident via drivers that are implemented to the Open Systems Development Lab Carrier Grade Linux Working Group (OSDL CGL WG) Driver Hardening APIs)
- Open source project information is located at https://sourceforge.net/projects/resourcemntrd/)

Multi-Router Traffic Grapher§ (MRTG)

- Graphs Ethernet network device statistics derived from the standard Internet Engineering Task Force (IETF) SNMP Resource Monitoring Management Information Block (RMON MIB)
- From the MRTG website:

  The Multi Router Traffic Grapher (MRTG) is a tool to monitor the traffic load on network-links. MRTG generates HTML pages containing graphical images which provide a LIVE visual representation of this traffic.
- Accesses data provided by SNMP subagents for the RMON MIB
- Open source project information is located at http://people.ee.ethz.ch/~oetiker/webtools/mrtg/
- Further investigation required to determine if MRTG can access the Resource Monitor MIB

KSysGuard§

- User interface for viewing system statistics; some monitoring and visual alarm capability; no event indication propagation capability
- From the KSysGuard website:

  KSysGuard is the KDE Task Manager and Performance Monitor. It features a client server architecture that allows monitoring of local as well as remote hosts. The graphical front end uses so called sensors to retrieve the information it displays.
- Is distributed with the KDE development environment
- Has sensors for `/proc` data (memory, task, CPU, etc.)

Open source project information is located at www.kde.org. Also, see http://docs.kde.org/2.2.2/kdebase/ksysguard/ for documentation on KSysGuard itself.

- Resource Monitor-based sensors could be used to extend coverage of the KSysGuard user interface application

MON§

- Described as a resource monitor; better categorized as a fault manager implementation; provides a plug-in architecture for event sensor and action plug-ins.
- From the MON home page:

mon is a general-purpose scheduler and alert management tool used for monitoring service availability and triggering alerts upon failure detection.

- Open source project information is located at http://ftp.kernel.org/software/mon/
- Would benefit from events generated by the Resource Monitor
  - A MON plug-in for Resource Monitor events would be useful
  - A MON plug-in for any POSIX 1003.25 log event would be useful

Performance Co-Pilot§ (PCP)

- Similar architecture to the Resource Monitor; features a collection daemon, plug-in metric access components, programmatic access for applications, and metric data persistence.
- Key differences between PCP and the Resource Monitor:
  - Scope of monitoring
    - PCP provides a framework for collating data from many servers in a network, especially beneficial for collating metrics gathered from members of a cluster.
    - Resource Monitor's scope is server-level only.
  - Event generation
    - PCP generates events from data collected at a central console or management server collection center.
    - Resource Monitor generates events locally on the managed server. Fault management and/or cluster manager applications can see events and take action locally, without involving remote management applications.
  - Data persistence
    - PCP persists data over the network to a central collection repository. This supports a key feature of PCP which is "retrospective analysis." PCP has agents that observe data over time and as the data is collected, determines fault conditions and invokes user configured actions.
    - Resource Monitor provides a plug-in interface for collecting statistic samples. The plug-in can persist the data in any means it chooses: binary, character, formatted, and so forth. The system integrator must provide a mechanism for forwarding collected data to a central site.
- From the PCP home page:

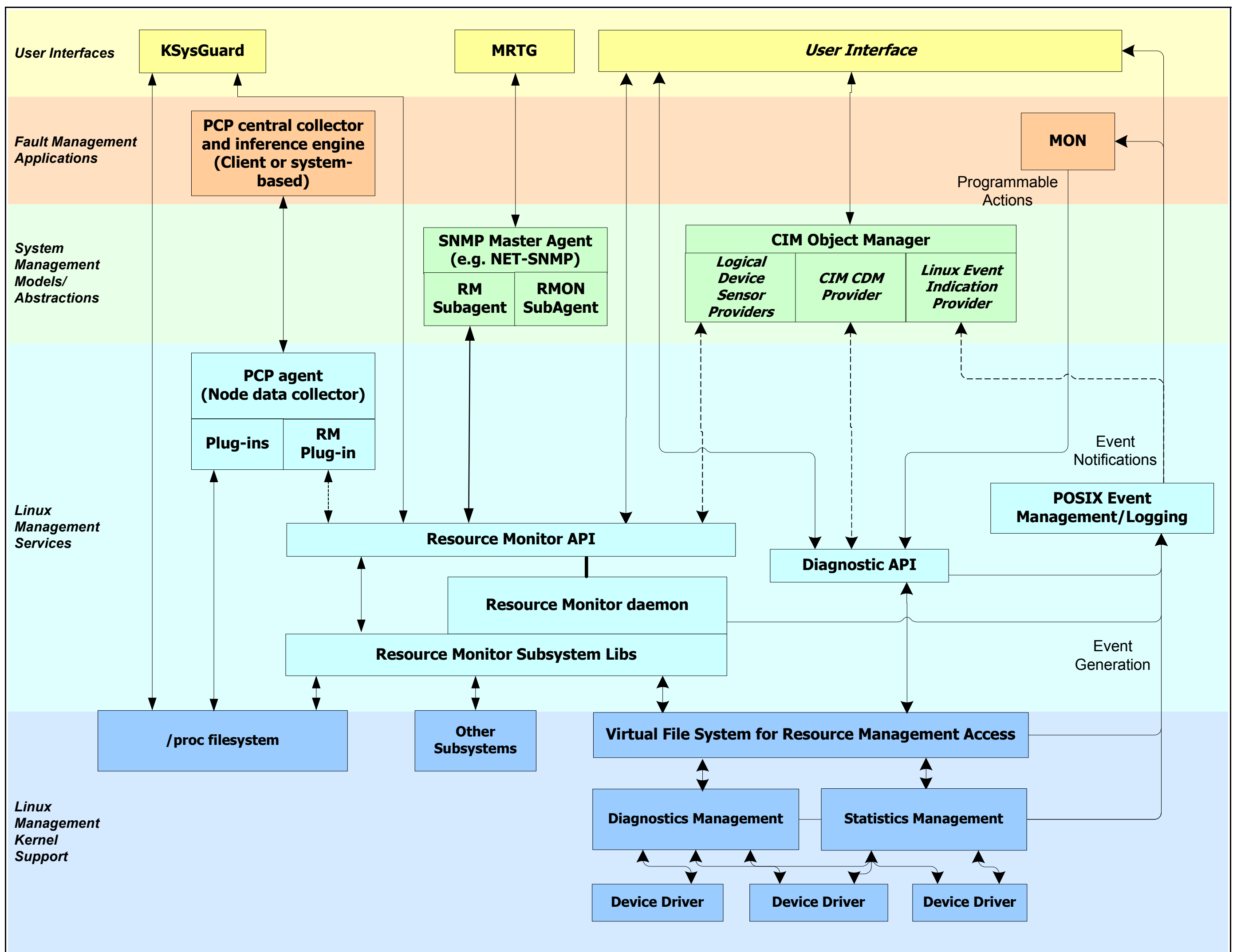
  Performance Co-Pilot (PCP) is a framework [that] supports system-level performance monitoring and performance management.

  The services offered by PCP are especially attractive for those tackling harder system-level performance problems. For example this may involve a transient performance degradation, or correlating end-user quality of service with platform activity, or diagnosing some complex interaction between resource demands on a single system, or management of performance on large systems with lots of "moving parts".

  The distributed PCP architecture makes it especially useful for those seeking centralized monitoring of distributed processing (e.g. in a cluster or webserver farm environment), especially where a large number [of] hosts are involved.

- PCP began life as a Silicon Graphics, Inc. (SGI) product. In February 2000, SGI released the infrastructure portion of the framework as open source under the GNU Public License (GPL), maintaining the open source on an SGI site (http://oss.sgi.com/projects/pcp/).

Figure 2 Linux system management architecture and components



These applications all form part of a complete system fault management solution. Figure 2 illustrates where each of these components would be used in a fault management system.

Implementation

Given the requirements reviewed in the “Analysis” section of this paper, the recommended implementation for monitoring critical resources is the Resource Monitor. The Resource Monitor provides the following capabilities:

- Monitors resources accessed through
  - Drivers and other loadable modules
  - Kernel
  - User-space applications
- Generates events via the POSIX Standard 1003.25 Event Logging/Management Facility (`evlog`).
- Supports multiple applications simultaneously monitoring multiple resources.
- Allows new resource types to be added dynamically.
- Allows multiple consumers and monitors:
  - Each consumer application can access any statistic.
  - Each consumer application can create multiple, independent monitors for any single statistic.
- Persists sampled statistic data through data capture plug-ins.

Resource Monitor Architecture

The Resource Monitor architecture recognizes the following elements:

- **Consumer applications:** software components that use the Resource Monitor application interfaces to monitor system resources. Consumer applications may be user-driven interfaces, autonomous fault managers, or network management agents.
- **Resources:** any identifiable, manageable parts of a system. For Resource Monitor, a resource is any entity which presents measurable attributes through a software interface.
- **Subsystems:** software components that represent a group of like resources. Each resource in a given subsystem is described by the same set of statistics.
- **Statistics:** measurable attributes of a resource.
- **Monitors:** active components that are configured to observe the value of a statistic at regular intervals and report event data based on a test condition for the observed value.
- **Events:** indications that a monitor generates when a statistic value meets a test condition of the monitor.
- **Data persistence libraries:** software components that persist sampled statistic values in a format expected by a class of applications.

Figure 3 Resource Monitor architecture

The Resource Monitor is comprised of the following components, as shown in Figure 3:

- **User interfaces**
  - Resource Monitor libraries, C++ and C libraries which provide programmatic access and control of statistics and monitors exposed through the Resource Monitor.
  - `rmxml`, a Linux utility that allows the same access and control of monitoring for shell script applications using extensible markup language (XML) syntax for identifying statistics and for establishing monitor parameters.
- **Resource Monitor daemon**
  - A Linux daemon process that receives commands from applications and either dispatches the monitoring requests to lower-level software or creates and maintains monitors on behalf of the application. The commands are received via a socket interface supported by the Resource Monitor library.
  - The daemon manages two classes of objects, statistics and monitors.

- **Subsystem libraries**
    - Subsystem libraries are dynamically linked libraries that provide resource-specific implementations for statistic and monitor classes providing access for the Resource Monitor daemon to any resource in the system.
    - Subsystem libraries implement the Resource Monitor daemon's access to statistics and monitors located in all system components, drivers, kernel modules, and applications.
- **Event Manager**
    - The Resource Monitor conforms to the event capture and event subscriber interfaces of the POSIX 1003.25 event management definition and by default uses the implementation "`evlog`". The Resource Monitor daemon and subsystem drivers that have been hardened to the CGL WG Driver Hardening specifications both log events via the "`evlog`" facility. The POSIX event logging interfaces allow applications to create persistent event queries that will call back to the application when events matching the query are processed.
- **Data Persistence Library**
    - The Resource Monitor exposes a plug-in interface that allows data collected from subsystem library-accessed statistics to be persisted for offline use. For example, a data persistence library could write the data in a compact, binary format to a disk drive-based file or could forward the data over a network to the network operations center (NOC) for offline analysis.

The following sections provide details about the internal design of the Resource Monitor daemon and the interfaces between components.

Resource Monitor Daemon Design

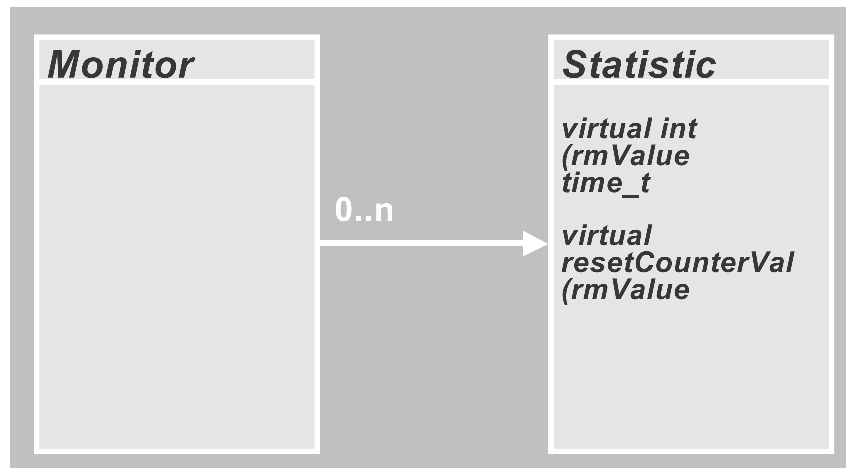
Daemon Object Model

The Resource Monitor daemon is an object-oriented service. The core object classes are *monitor* and *statistic*. For each statistic instantiated by the Resource Monitor daemon, the daemon can associate multiple monitors as requested by applications.

For monitors that the application requests to be maintained in user space, the Resource Monitor daemon coordinates sampling of the associated statistics such that a statistic is only sampled once per sampling interval regardless of the number of associated monitors.

For monitors that the application requests to be maintained "in-line", the Resource Monitor daemon uses the monitor class as a record of the settings and status of the in-line monitor.

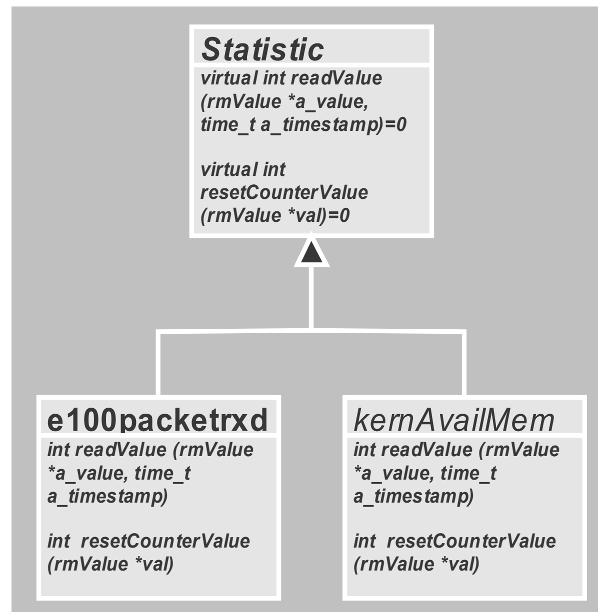
Figure 4 Resource Monitor daemon object model



Class Hierarchy

To allow the monitoring facility to be extended for any type of resource type or statistic type, the Resource Monitor daemon defines the *statistic* class as a virtual class. The methods for retrieving the statistic values are defined as pure virtual functions, such that implementers of the class must provide an implementation for the function. Statistics are implemented by subsystem libraries. Subsystem libraries are dynamically-linked libraries that provide implementations for statistic and monitor classes used by the Resource Monitor daemon.

Figure 5 Resource Monitor daemon class hierarchy for statistic



Application Interfaces

The application, or consumer, interfaces are defined in the Resource Monitor header files. The following sections highlight portions of the user interface. Consult the source site for the full interface and latest revisions.

Browsing

The following structures are used by the user application to navigate and discover manageable objects in the Resource Monitor's scope.

```
    size_t getSubsystemCount();
    int getAvailableSubsystems(rmSubsystemInfo *buffer, const size_t size);
    int getActiveResources(const RMuid guid, rmResourceInfo *buffer, const
size_t size);
    int getAvailableStatistics(const RMuid guid, rmStatisticInfo *buffer,
const size_t size);
    size_t getMonitorCount();
    int getMonitors(rmMonitorInfo *buffer, size_t *size);
    int getSubsystemInfo(const RMuid Subsystem, rmSubsystemInfo *buffer);
    int getResourceInfo(const RMuid Subsystem, const rmID id, rmResourceInfo
*buffer);
    int getStatisticInfo(const RMuid Subsystem, const rmID id,
rmStatisticInfo *buffer);
    rmString getSubsystemDescription(const RMuid Subsystem, const
rmDescriptions which);
```

```
    rmString getResourceDescription(const RMuid Subsystem, const rmID id,
const rmDescriptions which);
    rmString getStatisticDescription(const RMuid Subsystem, const rmID id,
const rmDescriptions which);
   int getResourceID(const RMuid Subsystem, const rmString shortdescription,
rmID *id);
    int getStatisticID(const RMuid Subsystem, const rmString
shortdescription, rmID *id);
```

Statistic

The following structures are used by the user application to access resource statistics.

```
    int getCurrentValue(const rmStatisticKey id, rmValue *value);
    int getUpperBound(const rmStatisticKey id, rmValue *value);
    int resetCounterStatistic(const rmStatisticKey id, const rmValue
*value);
```

Monitor

The following structures are used by the user application to create monitors and manage their activation.

```
    rmHandle createMonitor(const rmMonitorConfiguration &config,
rmMonitorControl *control);
    rmHandle accessMonitor(const RMuid MonitorId);
    int deleteMonitor(const rmHandle handle);
    int setMonitorControl(const rmHandle handle, const rmMonitorControl
&control);
    int getMonitorControl(const rmHandle handle, rmMonitorControl *control);
    int setMonitorConfiguration(const rmHandle handle, const
rmMonitorConfiguration &config);
    int getMonitorConfiguration(const rmHandle handle,
rmMonitorConfiguration *config);
    int startMonitor(const rmHandle handle);
    int stopMonitor(const rmHandle handle);
    int resetMonitor(const rmHandle handle);
    int pauseNotification(const rmHandle handle);
    int resetNotification(const rmHandle handle);
    int getMonitorState(const rmHandle handle, rmMonitorState *status);
    int getMonitorInfo(const rmHandle handle, rmMonitorInfo *buffer);
    rmString getMonitorDescription(const rmHandle handle, const
rmDescriptions which);
    int setMonitorDescription(const rmHandle handle, const rmDescriptions
which, const rmString description);
```

Monitor Configuration and Control Structures

The following structures are used by the user application to configure monitor test conditions and to control monitor event generation.

```
typedef struct
{
    /// used to switch on the typeConfiguration union
    enum rmMonitorType     monitorType;
    /// the triplet that identifies a specific instance of a statistic.
    rmStatisticKey         statisticKey;
```

```
    /// A statistic that provides an upperbound, normally gauges, the
percentage of
    /// the statistic is monitored when the monitor transformation is \a
rmPercent.
    /// With \a rmPercent, monitors can not be created for statistics when
no upper bound is available.
    /// The valid threshold values are 0 .. 100 when the \a rmPercent
transform is used.
    /// \a rmChange monitors the value of the change in a statistic reading.
    /// Since the change can be positive or negative,
    /// the \a thresholdValue is always assumed to be a signed.
    /// \a rmChange monitors the largest increase in a statistic value in
rmHighWatermark monitors.
    /// \a rmChange monitors the largest decrease in a statistic value in
rmLowWatermark monitors.
    /// \note Since statistic transforms effectively turn any statistic into
a gauge, the Leaky Bucket monitor
    /// should not be used with transforms because it only works predictably
with counters. Data capture
    /// only captures the raw reading, not the transformed value.
    enum rmStatisticTransform              statisticTransform;
    /// monitor type specific configuration parameters.
    union {
            rmThresholdConfiguration       threshold;
            rmWatermarkConfiguration       watermark;
            rmLeakyBucketConfiguration     leakyBucket;
            // for configuration of future monitor types
    } typeConfiguration;
} rmMonitorConfiguration;

typedef struct
{
    /// used to switch on the typeControl union.
    enum rmMonitorType     monitorType;
    /// specifies the inline or daemon location to use.
    enum rmMonitorLocation location;
    /// The monitor uuid will be part of an event record to uniquely
identify this
    /// monitor. The resource monitoring facility generates this unique ID
when
    /// a monitor is created if the uid is clear and is returned when
    /// ResourceMonitor::rmCreateMonitor is successful,
    /// else the provided uuid is used.
    /// \sa clearMonitorUID setMonitorUID
    rmUID                                  uid;
    /// the resource monitoring facility registers data capture libraries
using unique ID's.
    /// Each library provides a different storage method.
    /// Clearing this field indicates that no data capture will be done for
this monitor
    /// \sa clearDataCapture setDataCapture TextDataCapture
```

```
    /// \note Data capture only captures the raw reading, not the
  transformed value.
    rmUID                           dataCapture;
    /// seconds to monitor once started. The daemon automatically stops
  monitoring at the end of the interval.
    rmTimeInterval monitoringInterval;
    /// seconds between each sample taken by the monitor: 60 = 1
  sample/minute, etc.
    rmTimeInterval         monitoringRate;
    /// microseconds between each sample taken by the monitor added to \a
  monitoringRate
    /// Daemon monitors only use \a monitoringRate.
    rmMicroTimeInterval  microMonitoringRate;
    /// monitor type specific control parameters.
    union {
            rmThresholdControl threshold;
            rmLeakyBucketControl leakyBucket;
            // for control of future monitor types
    } typeControl;
} rmMonitorControl;
```

Subsystem Interfaces

Subsystem libraries are implemented as dynamically linkable C++ libraries (`.so` suffix). The subsystem libraries may be installed anywhere in the system. An entry in the file `/etc/opt/resourcemon/rmtab` is needed to indicate the location of the library location to the Resource Monitor daemon.

To expose a single subsystem to the daemon, the library provides an implementation for the virtual class *ISubsystemMonitor.* The Resource Monitor daemon opens all subsystem libraries found in `/etc/opt/resourcemon/rmtab` and constructs an instance of the *ISubsystemMonitor* class. In the constructor, the subsystem library developer uses the Resource Monitor daemon's static registration methods (*IRMRegistration*) to register any statistics, monitors, and/or resources provided by the subsystem library.

Process Flows

Figures 6 and 7 illustrate how a user application uses the Resource Monitor to create monitors and receive monitoring events.

Figure 6 Consumer application creates and starts a daemon-based monitor

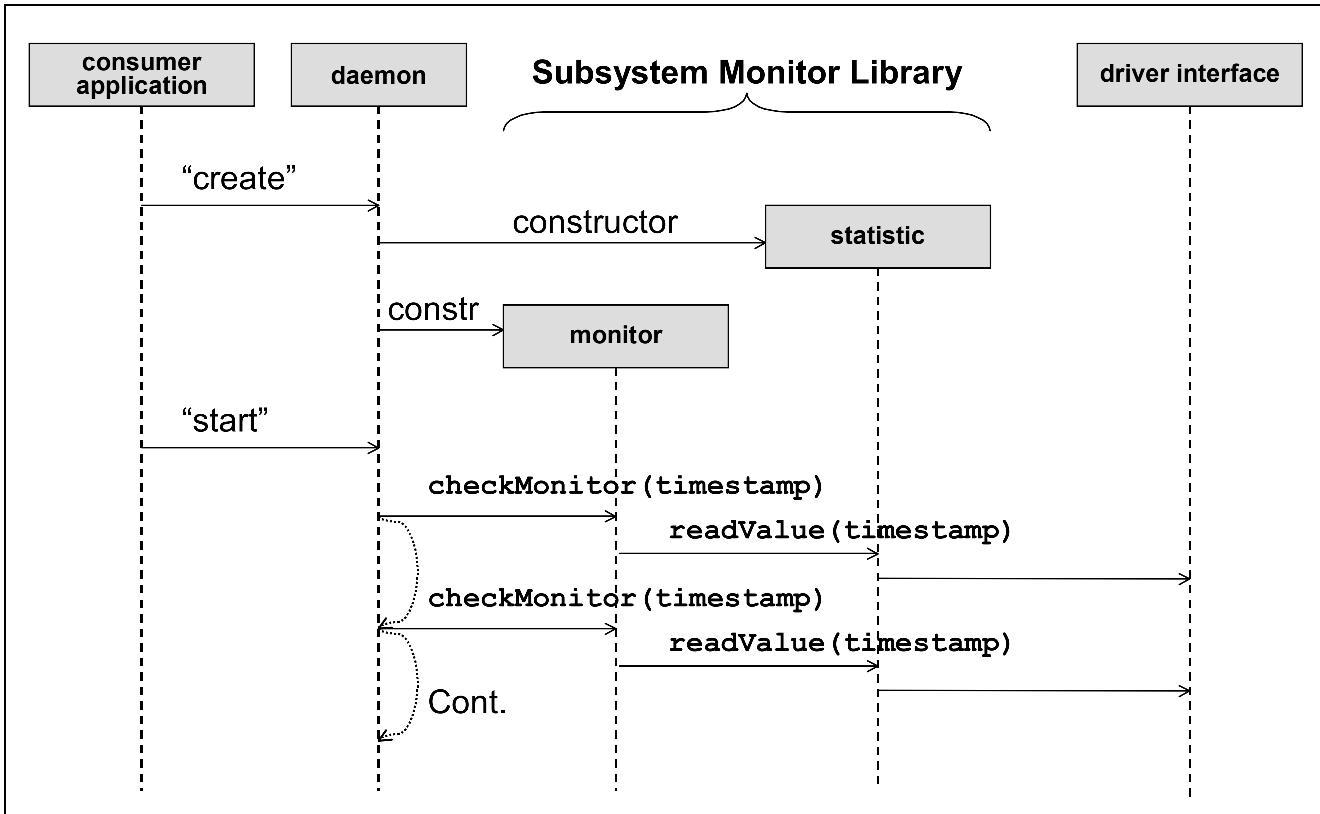


Figure 7 A threshold event is generated and received by an application

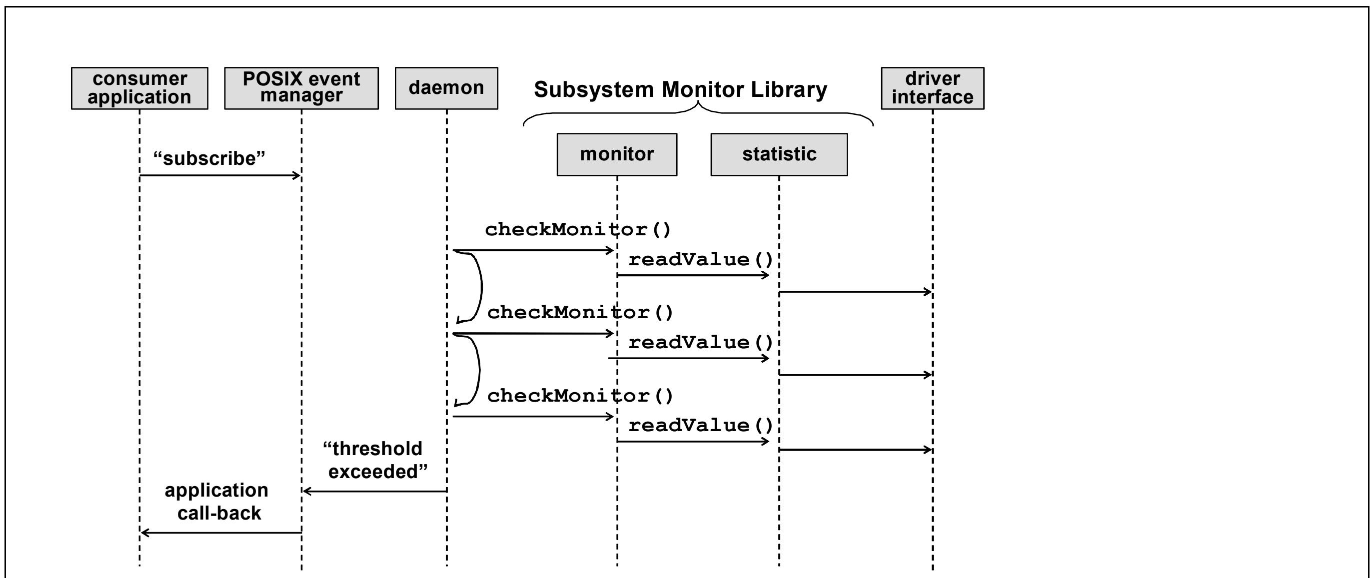
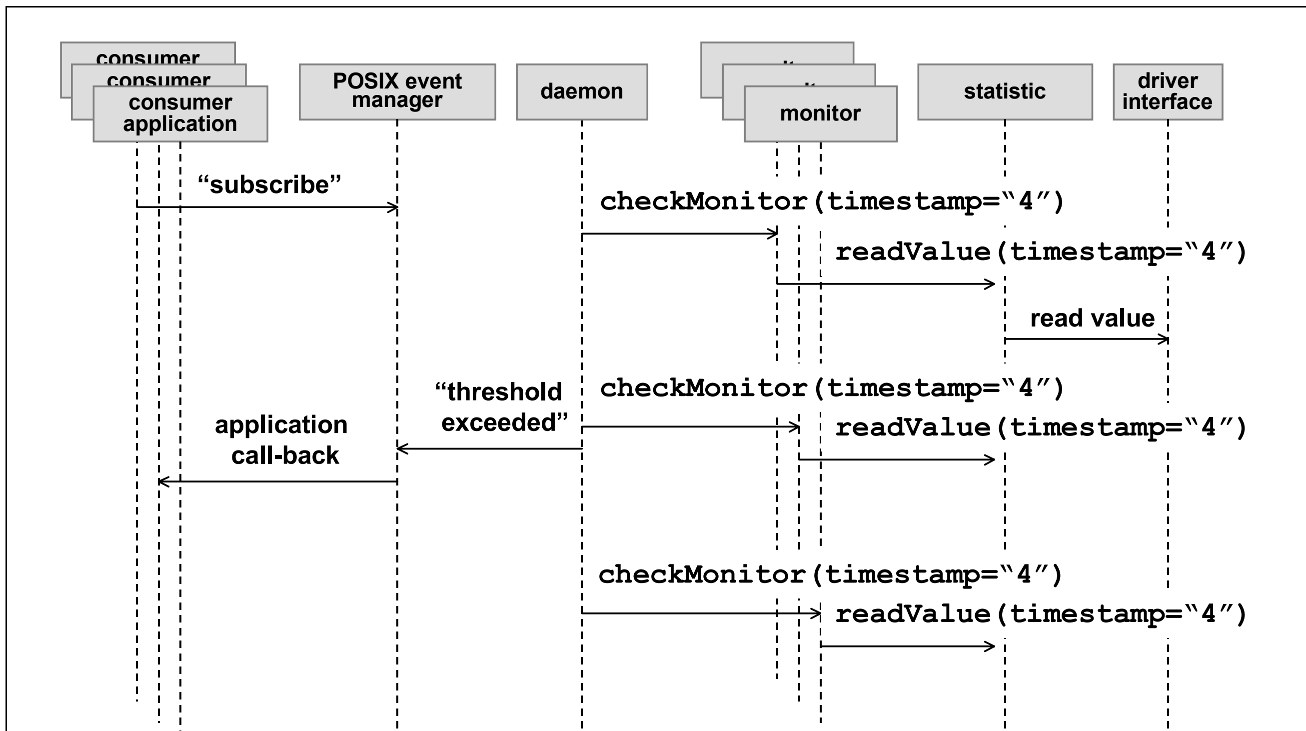


Figure 8 shows how multiple applications can create monitors with different configurations for the same statistic.

Figure 8 Multiple applications monitoring a single statistic



Consumer Applications

An SNMP subagent implementation is available that exposes the Resource Monitor's capabilities via an SNMP MIB.

TODOs

As with any open-source project, there are enhancements that are needed but not yet developed. The following is a list of components that would be valuable additions to Resource Monitor:

- **Network Management Framework integration**
  - **CIM Sensor Providers** would expose the monitoring capabilities to applications using the CIM framework to access management capabilities. The CIM classes for sensors and indications were used as models for developing the Resource Monitor daemon class infrastructure.
- **New second-order statistics and monitor types**
- **New subsystem monitoring libraries** for resource types not accessible today
  - TCP/IP stack subsystem library
  - Other protocol stack subsystem libraries
- **Integration with existing open-source Linux management tools**
  - KSysGuard – link Resource Monitor monitoring capability to KSysGuard visual objects
  - MON – develop MON event plug-ins for `evlog`, in general, and Resource Monitor monitoring events, specifically
  - PCP – develop a node data collector plug-in that utilizes Resource Monitor subsystem libraries for statistic exposure and data collection

Conclusion

The Resource Monitor is flexible, extensible, and practical. It allows the user to choose where monitoring will occur, either in driver or kernel time or in user/application time. The Resource Monitor also provides many choices for application development—C++, C or shell script. In addition, it can be extended to support new resource types via a dynamic library plug-in interface and supports all drivers hardened to the CGL WG driver hardening specifications.

Appendix A: References

Device Driver Hardening Design Specification, June 2002

Linux Event Logging for Enterprise-Class Systems, Open Source Project “evlog”: http://evlog.sourceforge.net/

NET-SNMP, Open Source Project for SNMP Monitoring: http://net-snmp.sourceforge.net/

OSDL Carrier Grade Linux Working Group: http://www.osdl.org/projects/cgl/

CGL Developer Sites: http://developer.osdl.org/

Distributed Management Task Force, Inc.: http://www.dmtf.org/

The Internet Engineering Task Force: http://www.ietf.org/

Appendix B: Abbreviations, Acronyms and Definitions

Abbreviation	Description
API	Application Programming Interface
CDM	Common Diagnostics Model, a design for providing common access methods for diagnostics across CIM managed elements
CIM	Common Information Model, Distributed Management Task Force system management object model standard
CGL	Carrier Grade Linux
CGL WG	Carrier Grade Linux Working Group
CPU	Central Processing Unit
DMI	Desktop Management Interface, DMTF standard for management
IETF	Internet Engineering Task Force
lib	library, an object component that is linked into an application to provide an interface or service access
MIB	Management Information Block, an abstraction of management data access used in SNMP
northbound	Traditional description used by carriers to describe the link from a network element (equipment) to a network operations center (NOC). A northbound link ties the network element into the management plane of the network architecture.
OSDL	Open Systems Development Lab
POSIX	Portable Operating System Interfaces standard
RMON	SNMP Remote Monitoring MIB
SNMP	Simple Network Management Protocol (Internet Engineering Task Force standard)
XML	eXtensible Markup Language