pcp Documentation

Performance Co-Pilot

Mar 16, 2021

GUIDES

1 About User's and Administrator's Guide **3**
- 1.1 What This Guide Contains . . . 3
- 1.2 Audience for This Guide . . . 4
- 1.3 Related Resources . . . 4
- 1.4 Man Pages . . . 4
- 1.5 Web Site . . . 5
- 1.6 Conventions . . . 5
- 1.7 Reader Comments . . . 5

2 About Programmer's Guide **7**
- 2.1 What This Guide Contains . . . 7
- 2.2 Audience for This Guide . . . 8
- 2.3 Related Resources . . . 8
- 2.4 Man Pages . . . 8
- 2.5 Web Site . . . 9
- 2.6 Conventions . . . 9
- 2.7 Reader Comments . . . 9

3 Introduction to PCP **11**
- 3.1 Objectives . . . 12
- 3.2 Conceptual Foundations . . . 15
- 3.3 Overview of Component Software . . . 20

4 Installing and Configuring Performance Co-Pilot **27**
- 4.1 Product Structure . . . 28
- 4.2 Performance Metrics Collection Daemon (PMCD) . . . 28
- 4.3 Managing Optional PMDAs . . . 34
- 4.4 Troubleshooting . . . 36

5 Common Conventions and Arguments **41**
- 5.1 Alternate Metrics Source Options . . . 42
- 5.2 General PCP Tool Options . . . 43
- 5.3 Time Duration and Control . . . 44
- 5.4 PCP Environment Variables . . . 46
- 5.5 Running PCP Tools through a Firewall . . . 48
- 5.6 Transient Problems with Performance Metric Values . . . 48

6 Monitoring System Performance **51**
- 6.1 The pmstat Command . . . 52
- 6.2 The pmrep Command . . . 53
- 6.3 The pmval Command . . . 53

6.4 The pminfo Command . . . 55
6.5 The pmstore Command . . . 58

7 Performance Metrics Inference Engine 61
7.1 Introduction to pmie . . . 62
7.2 Basic pmie Usage . . . 64
7.3 Specification Language for pmie . . . 67
7.4 pmie Examples . . . 77
7.5 Developing and Debugging pmie Rules . . . 79
7.6 Caveats and Notes on pmie . . . 79
7.7 Creating pmie Rules with pmieconf . . . 80
7.8 Management of pmie Processes . . . 83

8 Archive Logging 87
8.1 Introduction to Archive Logging . . . 88
8.2 Using Archive Logs with Performance Tools . . . 90
8.3 Cookbook for Archive Logging . . . 93
8.4 Other Archive Logging Features and Services . . . 96
8.5 Archive Logging Troubleshooting . . . 99

9 Performance Co-Pilot Deployment Strategies 103
9.1 Basic Deployment . . . 104
9.2 PCP Collector Deployment . . . 105
9.3 PCP Archive Logger Deployment . . . 106
9.4 PCP Inference Engine Deployment . . . 107

10 Customizing and Extending PCP Services 111
10.1 PMDA Customization . . . 112
10.2 PCP Tool Customization . . . 114
10.3 PMNS Management . . . 116
10.4 PMDA Development . . . 118
10.5 PCP Tool Development . . . 118

11 Fast, Scalable Time Series Querying - pmseries 119
11.1 Introduction to pmseries . . . 120
11.2 Timeseries Queries . . . 121
11.3 Metadata Qualifiers and Metadata Operators . . . 121
11.4 Time Specification . . . 122
11.5 Expressions . . . 123
11.6 Timeseries Options . . . 126
11.7 PCP Environment . . . 130
11.8 PCP Grafana Plugin . . . 131

12 Programming Performance Co-Pilot 133
12.1 PCP Architecture . . . 134
12.2 Overview of Component Software . . . 135
12.3 PMDA Development . . . 136
12.4 Client Development and PMAPI . . . 138
12.5 Library Reentrancy and Threaded Applications . . . 138

13 Writing A PMDA 139
13.1 Implementing a PMDA . . . 140
13.2 PMDA Architecture . . . 141
13.3 Domains, Metrics, Instances and Labels . . . 144
13.4 Other Issues . . . 154

13.5 PMDA Interface . . . 159
13.6 Initializing a PMDA . . . 166
13.7 Testing and Debugging a PMDA . . . 170
13.8 Integration of a PMDA . . . 172

14 PMAPI–The Performance Metrics API **175**
14.1 Naming and Identifying Performance Metrics . . . 178
14.2 Performance Metric Instances . . . 178
14.3 Current PMAPI Context . . . 180
14.4 Performance Metric Descriptions . . . 180
14.5 Performance Metrics Values . . . 183
14.6 Performance Event Metrics . . . 185
14.7 PMAPI Programming Style and Interaction . . . 189
14.8 PMAPI Procedural Interface . . . 190
14.9 PMAPI Programming Issues and Examples . . . 222

15 Instrumenting Applications **227**
15.1 Application and Performance Co-Pilot Relationship . . . 228
15.2 Performance Instrumentation and Sampling . . . 229
15.3 MMV PMDA Design . . . 229
15.4 Memory Mapped Values API . . . 229
15.5 Performance Instrumentation and Tracing . . . 234
15.6 Trace PMDA Design . . . 234
15.7 Trace API . . . 237

Performance Co-Pilot (PCP) provides a framework and services to support system-level performance monitoring and management. It presents a unifying abstraction for all of the performance data in a system, and many tools for interrogating, retrieving and processing that data.

PCP is a feature-rich, mature, extensible, cross-platform toolkit supporting both live and retrospective analysis. The distributed PCP architecture makes it especially useful for those seeking centralized monitoring of distributed processing.

Table of Contents

- *About User's and Administrator's Guide*
- *About Programmer's Guide*
- REST API Guide

CHAPTER

ONE

ABOUT USER'S AND ADMINISTRATOR'S GUIDE

Contents

- *About User's and Administrator's Guide*
  - *What This Guide Contains*
  - *Audience for This Guide*
  - *Related Resources*
  - *Man Pages*
  - *Web Site*
  - *Conventions*
  - *Reader Comments*

This guide describes the Performance Co-Pilot (PCP) performance analysis toolkit. PCP provides a systems-level suite of tools that cooperate to deliver distributed performance monitoring and performance management services spanning hardware platforms, operating systems, service layers, database internals, user applications and distributed architectures.

PCP is a cross-platform, open source software package - customizations, extensions, source code inspection, and tinkering in general is actively encouraged.

"About User's and Administrator's Guide" includes short descriptions of the chapters in this book, directs you to additional sources of information, and explains typographical conventions.

1.1 What This Guide Contains

This guide contains the following chapters:

Chapter 1, *Introduction to PCP*, provides an introduction, a brief overview of the software components, and conceptual foundations of the PCP software.

Chapter 2, *Installing and Configuring Performance Co-Pilot*, describes the basic installation and configuration steps necessary to get PCP running on your systems.

Chapter 3, *Common Conventions and Arguments*, describes the user interface components that are common to most of the text-based utilities that make up the monitor portion of PCP.

Chapter 4, *Monitoring System Performance*, describes the performance monitoring tools available in Performance Co-Pilot (PCP).

Chapter 5, *Performance Metrics Inference Engine*, describes the Performance Metrics Inference Engine (pmie) tool that provides automated monitoring of, and reasoning about, system performance within the PCP framework.

Chapter 6, *Archive Logging*, covers the PCP services and utilities that support archive logging for capturing accurate historical performance records.

Chapter 7, *Performance Co-Pilot Deployment Strategies*, presents the various options for deploying PCP functionality across cooperating systems.

Chapter 8, *Customizing and Extending PCP Services*, describes the procedures necessary to ensure that the PCP configuration is customized in ways that maximize the coverage and quality of performance monitoring and management services.

Chapter 9, *Fast, Scalable Time Series Querying - pmseries*, describes the concepts of pmseries and lays out the path to the Performance Co-Pilot (PCP) Grafana Plugin.

1.2 Audience for This Guide

This guide is written for the system administrator or performance analyst who is directly using and administering PCP applications.

1.3 Related Resources

The *Performance Co-Pilot Programmer's Guide*, a companion document to the *Performance Co-Pilot User's and Administrator's Guide*, is intended for developers who want to use the PCP framework and services for exporting additional collections of performance metrics, or for delivering new or customized applications to enhance performance management.

The *Performance Co-Pilot Tutorials and Case Studies* provides a series of real-world examples of using various PCP tools, and lessons learned from deploying the toolkit in production environments. It serves to provide reinforcement of the general concepts discussed in the other two books with additional case studies, and in some cases very detailed discussion of specifics of individual tools.

Additional resources include man pages and the project web site.

1.4 Man Pages

The operating system man pages provide concise reference information on the use of commands, subroutines, and system resources. There is usually a man page for each PCP command or subroutine. To see a list of all the PCP man pages, start from the following command:

```
man PCPIntro
```

Each man page usually has a "SEE ALSO" section, linking to other, related entries.

To see a particular man page, supply its name to the **man** command, for example:

```
man pcp
```

The man pages are arranged in different sections - user commands, programming interfaces, and so on. For a complete list of manual sections on a platform enter the command:

```
man man
```

When referring to man pages, this guide follows a standard convention: the section number in parentheses follows the item. For example, **pminfo(1)** refers to the man page in section 1 for the pminfo command.

1.5 Web Site

The following web site is accessible to everyone:

URL : https://pcp.io

PCP is open source software released under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL)

1.6 Conventions

The following conventions are used throughout this document:

Convention	Meaning
`${PCP_VARIABLE}`	A brace-enclosed all-capital-letters syntax indicates a variable that has been sourced from the global `${PCP_DIR}/etc/pcp.conf` file. These special variables indicate parameters that affect all PCP commands, and are likely to be different between platforms.
command	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
ALL CAPS	All capital letters denote environment variables, operator names, directives, defined constants, and macros in C programs.
()	Parentheses that follow function names surround function arguments or are empty if the function has no arguments; parentheses that follow commands surround man page section numbers.

1.7 Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact the PCP maintainers using either the email address or the web site listed earlier.

We value your comments and will respond to them promptly.

CHAPTER

TWO

ABOUT PROGRAMMER'S GUIDE

Contents

- *About Programmer's Guide*
  - *What This Guide Contains*
  - *Audience for This Guide*
  - *Related Resources*
  - *Man Pages*
  - *Web Site*
  - *Conventions*
  - *Reader Comments*

This guide describes how to program the Performance Co-Pilot (PCP) performance analysis toolkit. PCP provides a systems-level suite of tools that cooperate to deliver distributed performance monitoring and performance management services spanning hardware platforms, operating systems, service layers, database internals, user applications and distributed architectures.

PCP is an open source, cross-platform software package - customizations, extensions, source code inspection, and tinkering in general is actively encouraged.

"About Programmer's Guide" includes short descriptions of the chapters in this book, directs you to additional sources of information, and explains typographical conventions.

2.1 What This Guide Contains

This guide contains the following chapters:

Chapter 1, *Programming Performance Co-Pilot*, contains a thumbnail sketch of how to program the various PCP components.

Chapter 2, *Writing A PMDA*, describes how to write Performance Metrics Domain Agents (PMDAs) for PCP.

Chapter 3, *PMAPI–The Performance Metrics API*, describes the interface that allows you to design custom performance monitoring tools.

Chapter 4, *Instrumenting Applications*, introduces techniques, tools and interfaces to assist with exporting performance data from within applications.

2.2 Audience for This Guide

The guide describes the programming interfaces to Performance Co-Pilot (PCP) for the following intended audience:

- Performance analysts or system administrators who want to extend or customize performance monitoring tools available with PCP
- Developers who wish to integrate performance data from within their applications into the PCP framework

This book is written for those who are competent with the C programming language, the UNIX or the Linux operating systems, and the target domain from which the desired performance metrics are to be extracted. Familiarity with the PCP tool suite is assumed.

2.3 Related Resources

The *Performance Co-Pilot User's and Administrator's Guide* is a companion document to the *Performance Co-Pilot Programmer's Guide*, and is intended for system administrators and performance analysts who are directly using and administering PCP installations.

The *Performance Co-Pilot Tutorials and Case Studies* provides a series of real-world examples of using various PCP tools, and lessons learned from deploying the toolkit in production environments. It serves to provide reinforcement of the general concepts discussed in the other two books with additional case studies, and in some cases very detailed discussion of specifics of individual tools.

Additional resources include man pages and the project web site.

2.4 Man Pages

The operating system man pages provide concise reference information on the use of commands, subroutines, and system resources. There is usually a man page for each PCP command or subroutine. To see a list of all the PCP man pages, start from the following command:

```
man PCPIntro
```

Each man page usually has a "SEE ALSO" section, linking to other, related entries.

To see a particular man page, supply its name to the **man** command, for example:

```
man pcp
```

The man pages are arranged in different sections separating commands, programming interfaces, and so on. For a complete list of manual sections on a platform enter the command:

```
man man
```

When referring to man pages, this guide follows a standard convention: the section number in parentheses follows the item. For example, **pminfo(1)** refers to the man page in section 1 for the **pminfo** command.

2.5 Web Site

The following web site is accessible to everyone:

URL : https://pcp.io

PCP is open source software released under the GNU General Public License (GPL) and GNU Lesser General Public License (LGPL)

2.6 Conventions

The following conventions are used throughout this document:

Convention	Meaning
`${PCP_VARIABLE}`	A brace-enclosed all-capital-letters syntax indicates a variable that has been sourced from the global `${PCP_DIR}/etc/pcp.conf` file. These special variables indicate parameters that affect all PCP commands, and are likely to be different between platforms.
command	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
ALL CAPS	All capital letters denote environment variables, operator names, directives, defined constants, and macros in C programs.
()	Parentheses that follow function names surround function arguments or are empty if the function has no arguments; parentheses that follow commands surround man page section numbers.

2.7 Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact the PCP maintainers using either the email address or the web site listed earlier.

We value your comments and will respond to them promptly.

CHAPTER

THREE

INTRODUCTION TO PCP

Contents

- *Introduction to PCP*
  - *Objectives*
    - *PCP Target Usage*
    - *Empowering the PCP User*
    - *Unification of Performance Metric Domains*
    - *Uniform Naming and Access to Performance Metrics*
    - *PCP Distributed Operation*
    - *Dynamic Adaptation to Change*
    - *Logging and Retrospective Analysis*
    - *Automated Operational Support*
    - *PCP Extensibility*
    - *Metric Coverage*
  - *Conceptual Foundations*
    - *Performance Metrics*
    - *Performance Metric Instances*
    - *Current Metric Context*
    - *Sources of Performance Metrics and Their Domains*
    - *Distributed Collection*
    - *Performance Metrics Name Space*
      - *Performance Metrics Name Space Diagram*
    - *Descriptions for Performance Metrics*
    - *Values for Performance Metrics*
      - *Single-Valued Performance Metrics*
      - *Set-Valued Performance Metrics*
    - *Collector and Monitor Roles*

        * *Retrospective Sources of Performance Metrics*
        * *Product Extensibility*
    - *Overview of Component Software*
        * *Performance Monitoring and Visualization*
        * *Collecting, Transporting, and Archiving Performance Information*
        * *Operational and Infrastructure Support*
        * *Application and Agent Development*

This chapter provides an introduction to Performance Co-Pilot (PCP), an overview of its individual components, and conceptual information to help you use this software.

The following sections are included:

Section 1.1, "*Objectives*" covers the intended purposes of PCP.

Section 1.2, "*Conceptual Foundations*", discusses the design theories behind PCP.

Section 1.3, "*Overview of Component Software*", describes PCP tools and agents.

3.1 Objectives

Performance Co-Pilot (PCP) provides a range of services that may be used to monitor and manage system performance. These services are distributed and scalable to accommodate the most complex system configurations and performance problems.

3.1.1 PCP Target Usage

PCP is targeted at the performance analyst, benchmarker, capacity planner, developer, database administrator, or system administrator with an interest in overall system performance and a need to quickly isolate and understand performance behavior, resource utilization, activity levels, and bottlenecks in complex systems. Platforms that can benefit from this level of performance analysis include large servers, server clusters, or multiserver sites delivering Database Management Systems (DBMS), compute, Web, file, or video services.

3.1.2 Empowering the PCP User

To deal efficiently with the dynamic behavior of complex systems, performance analysts need to filter out noise from the overwhelming stream of performance data, and focus on exceptional scenarios. Visualization of current and historical performance data, and automated reasoning about performance data, effectively provide this filtering.

From the PCP end user's perspective, PCP presents an integrated suite of tools, user interfaces, and services that support real-time and retrospective performance analysis, with a bias towards eliminating mundane information and focusing attention on the exceptional and extraordinary performance behaviors. When this is done, the user can concentrate on in-depth analysis or target management procedures for those critical system performance problems.

3.1.3 Unification of Performance Metric Domains

At the lowest level, performance metrics are collected and managed in autonomous performance domains such as the operating system kernel, a DBMS, a layered service, or an end-user application. These domains feature a multitude of access control policies, access methods, data semantics, and multiversion support. All this detail is irrelevant to the developer or user of a performance monitoring tool, and is hidden by the PCP infrastructure.

Performance Metrics Domain Agents (PMDAs) within PCP encapsulate the knowledge about, and export performance information from, autonomous performance domains.

3.1.4 Uniform Naming and Access to Performance Metrics

Usability and extensibility of performance management tools mandate a single scheme for naming performance metrics. The set of defined names constitutes a Performance Metrics Name Space (PMNS). Within PCP, the PMNS is adaptive so it can be extended, reshaped, and pruned to meet the needs of particular applications and users.

PCP provides a single interface to name and retrieve values for all performance metrics, independently of their source or location.

3.1.5 PCP Distributed Operation

From a purely pragmatic viewpoint, a single workstation must be able to monitor the concurrent performance of multiple remote hosts. At the same time, a single host may be subject to monitoring from multiple remote workstations.

These requirements suggest a classic client-server architecture, which is exactly what PCP uses to provide concurrent and multiconnected access to performance metrics, independent of their host location.

3.1.6 Dynamic Adaptation to Change

Complex systems are subject to continual changes as network connections fail and are reestablished; nodes are taken out of service and rebooted; hardware is added and removed; and software is upgraded, installed, or removed. Often these changes are asynchronous and remote (perhaps in another geographic region or domain of administrative control).

The distributed nature of the PCP (and the modular fashion in which performance metrics domains can be installed, upgraded, and configured on different hosts) enables PCP to adapt concurrently to changes in the monitored system(s). Variations in the available performance metrics as a consequence of configuration changes are handled automatically and become visible to all clients as soon as the reconfigured host is rebooted or the responsible agent is restarted.

PCP also detects loss of client-server connections, and most clients support subsequent automated reconnection.

3.1.7 Logging and Retrospective Analysis

A range of tools is provided to support flexible, adaptive logging of performance metrics for archive, playback, remote diagnosis, and capacity planning. PCP archive logs may be accumulated either at the host being monitored, at a monitoring workstation, or both.

A universal replay mechanism, modeled on media controls, supports play, step, rewind, fast forward and variable speed processing of archived performance data. Replay for multiple archives, from multiple hosts, is facilitated by an archive aggregation concept.

Most PCP applications are able to process archive logs and real-time performance data with equal facility. Unification of real-time access and access to the archive logs, in conjunction with the media controls, provides powerful mechanisms for building performance tools and to review both current and historical performance data.

3.1.8 Automated Operational Support

For operational and production environments, PCP provides a framework with scripts to customize in order to automate the execution of ongoing tasks such as these:

- Centralized archive logging for multiple remote hosts
- Archive log rotation, consolidation, and culling
- Web-based publishing of charts showing snapshots of performance activity levels in the recent past
- Flexible alarm monitoring: parameterized rules to address common critical performance scenarios and facilities to customize and refine this monitoring
- Retrospective performance audits covering the recent past; for example, daily or weekly checks for performance regressions or quality of service problems

3.1.9 PCP Extensibility

PCP permits the integration of new performance metrics into the PMNS, the collection infrastructure, and the logging framework. The guiding principle is, "if it is important for monitoring system performance, and you can measure it, you can easily integrate it into the PCP framework."

For many PCP users, the most important performance metrics are not those already supported, but new performance metrics that characterize the essence of good or bad performance at their site, or within their particular application environment.

One example is an application that measures the round-trip time for a benign "probe" transaction against some mission-critical application.

For application developers, a library is provided to support easy-to-use insertion of trace and monitoring points within an application, and the automatic export of resultant performance data into the PCP framework. Other libraries and tools aid the development of customized and fully featured Performance Metrics Domain Agents (PMDAs).

Extensive source code examples are provided in the distribution, and by using the PCP toolkit and interfaces, these customized measures of performance or quality of service can be easily and seamlessly integrated into the PCP framework.

3.1.10 Metric Coverage

The core PCP modules support export of performance metrics that include kernel instrumentation, hardware instrumentation, process-level resource utilization, database and other system services instrumentation, and activity in the PCP collection infrastructure.

The supplied agents support thousands of distinct performance metrics, many of which can have multiple values, for example, per disk, per CPU, or per process.

3.2 Conceptual Foundations

The following sections provide a detailed overview of concepts that underpin Performance Co-Pilot (PCP).

3.2.1 Performance Metrics

Across all of the supported performance metric domains, there are a large number of performance metrics. Each metric has its own structure and semantics. PCP presents a uniform interface to these metrics, independent of the underlying metric data source.

The Performance Metrics Name Space (PMNS) provides a hierarchical classification of human-readable metric names, and a mapping from these external names to internal metric identifiers. See Section 1.2.6, "*Performance Metrics Name Space*", for a description of the PMNS.

3.2.2 Performance Metric Instances

When performance metric values are returned to a requesting application, there may be more than one value instance for a particular metric; for example, independent counts for each CPU, process, disk, or local filesystem. Internal instance identifiers correspond one to one with external (human-readable) descriptions of the members of an instance domain.

Transient performance metrics (such as per-process information) cause repeated requests for the same metric to return different numbers of values, or changes in the particular instance identifiers returned. These changes are expected and fully supported by the PCP infrastructure; however, metric instantiation is guaranteed to be valid only at the time of collection.

3.2.3 Current Metric Context

When performance metrics are retrieved, they are delivered in the context of a particular source of metrics, a point in time, and a profile of desired instances. This means that the application making the request has already negotiated to establish the context in which the request should be executed.

A metric source may be the current performance data from a particular host (a live or real-time source), or a set of archive logs of performance data collected by **pmlogger** at some distant host or at an earlier time (a retrospective or archive source).

By default, the collection time for a performance metric is the current time of day for real-time sources, or current point within an archive source. For archives, the collection time may be reset to an arbitrary time within the bounds of the set of archive logs.

3.2.4 Sources of Performance Metrics and Their Domains

Instrumentation for the purpose of performance monitoring typically consists of counts of activity or events, attribution of resource consumption, and service-time or response-time measures. This instrumentation may exist in one or more of the functional domains as shown in *Figure 1.1. Performance Metric Domains as Autonomous Collections of Data*.

Fig. 1: Figure 1.1. Performance Metric Domains as Autonomous Collections of Data

Each domain has an associated access method:

- The operating system kernel, including sub-system data structures - per-process resource consumption, network statistics, disk activity, or memory management instrumentation.

- A layered software service such as activity logs for a World Wide Web server or an email delivery server.
- An application program such as measured response time for a production application running a periodic and benign probe transaction (as often required in service level agreements), or rate of computation and throughput in jobs per minute for a batch stream.
- External equipment such as network routers and bridges.

For each domain, the set of performance metrics may be viewed as an abstract data type, with an associated set of methods that may be used to perform the following tasks:

- Interrogate the metadata that describes the syntax and semantics of the performance metrics
- Control (enable or disable) the collection of some or all of the metrics
- Extract instantiations (current values) for some or all of the metrics

We refer to each functional domain as a performance metrics domain and assume that domains are functionally, architecturally, and administratively independent and autonomous. Obviously the set of performance metrics domains available on any host is variable, and changes with time as software and hardware are installed and removed.

The number of performance metrics domains may be further enlarged in cluster-based or network-based configurations, where there is potentially an instance of each performance metrics domain on each node. Hence, the management of performance metrics domains must be both extensible at a particular host and distributed across a number of hosts.

Each performance metrics domain on a particular host must be assigned a unique Performance Metric Identifier (PMID). In practice, this means unique identifiers are assigned globally for each performance metrics domain type. For example, the same identifier would be used for the Apache Web Server performance metrics domain on all hosts.

3.2.5 Distributed Collection

The performance metrics collection architecture is distributed, in the sense that any performance tool may be executing remotely. However, a PMDA usually runs on the system for which it is collecting performance measurements. In most cases, connecting these tools together on the collector host is the responsibility of the PMCD process, as shown in *Figure 1.2. Process Structure for Distributed Operation*.

Fig. 2: Figure 1.2. Process Structure for Distributed Operation

The host running the monitoring tools does not require any collection tools, including **pmcd**, because all requests for metrics are sent to the **pmcd** process on the collector host. These requests are then forwarded to the appropriate PMDAs, which respond with metric descriptions, help text, and most importantly, metric values.

The connections between monitor clients and **pmcd** processes are managed in **libpcp**, below the PMAPI level; see the **pmapi(3)** man page. Connections between PMDAs and **pmcd** are managed by the PMDA routines; see the **pmda(3)** man page. There can be multiple monitor clients and multiple PMDAs on the one host, but normally there would be only one **pmcd** process.

3.2.6 Performance Metrics Name Space

Internally, each unique performance metric is identified by a Performance Metric Identifier (PMID) drawn from a universal set of identifiers, including some that are reserved for site-specific, application-specific, and customer-specific use.

An external name space - the Performance Metrics Name Space (PMNS) - maps from a hierarchy (or tree) of human-readable names to PMIDs.

Performance Metrics Name Space Diagram

Each node in the PMNS tree is assigned a label that must begin with an alphabet character, and be followed by zero or more alphanumeric characters or the underscore (_) character. The root node of the tree has the special label of **root**.

A metric name is formed by traversing the tree from the root to a leaf node with each node label on the path separated by a period. The common prefix **root.** is omitted from all names. For example, *Figure 1.3. Small Performance Metrics Name Space (PMNS)* shows the nodes in a small subsection of a PMNS.

Fig. 3: Figure 1.3. Small Performance Metrics Name Space (PMNS)

In this subsection, the following are valid names for performance metrics:

```
kernel.percpu.syscall
network.tcp.rcvpack
hw.router.recv.total_util
```

3.2.7 Descriptions for Performance Metrics

Through the various performance metric domains, the PCP must support a wide range of formats and semantics for performance metrics. This *metadata* describing the performance metrics includes the following:

- The internal identifier, Performance Metric Identifier (PMID), for the metric
- The format and encoding for the values of the metric, for example, an unsigned 32-bit integer or a string or a 64-bit IEEE format floating point number
- The semantics of the metric, particularly the interpretation of the values as free-running counters or instantaneous values
- The dimensionality of the values, in the dimensions of events, space, and time
- The scale of values; for example, bytes, kilobytes (KB), or megabytes (MB) for the space dimension
- An indication if the metric may have one or many associated values
- Short (and extended) help text describing the metric

For each metric, this metadata is defined within the associated PMDA, and PCP arranges for the information to be exported to performance tools that use the metadata when interpreting the values for each metric.

3.2.8 Values for Performance Metrics

The following sections describe two types of performance metrics, single-valued and set-valued.

Single-Valued Performance Metrics

Some performance metrics have a singular value within their performance metric domains. For example, available memory (or the total number of context switches) has only one value per performance metric domain, that is, one value per host. The metadata describing the metric makes this fact known to applications that process values for these metrics.

Set-Valued Performance Metrics

Some performance metrics have a set of values or instances in each implementing performance metric domain. For example, one value for each disk, one value for each process, one value for each CPU, or one value for each activation of a given application.

When a metric has multiple instances, the PCP framework does not pollute the Name Space with additional metric names; rather, a single metric may have an associated set of values. These multiple values are associated with the members of an *instance domain*, such that each instance has a unique instance identifier within the associated instance domain. For example, the "per CPU" instance domain may use the instance identifiers 0, 1, 2, 3, and so on to identify the configured processors in the system.

Internally, instance identifiers are encoded as binary values, but each performance metric domain also supports corresponding strings as external names for the instance identifiers, and these names are used at the user interface to the PCP utilities.

For example, the performance metric **disk.dev.total** counts I/O operations for each disk spindle, and the associated instance domain contains one member for each disk spindle. On a system with five specific disks, one value would be associated with each of the external and internal instance identifier pairs shown in Table 1.1. Sample Instance Identifiers for Disk Statistics.

Table 1.1. Sample Instance Identifiers for Disk Statistics

External Instance Identifier	Internal Instance Identifier
disk0	131329
disk1	131330
disk2	131331
disk3	131841
disk4	131842

Multiple performance metrics may be associated with a single instance domain.

Each performance metric domain may dynamically establish the instances within an instance domain. For example, there may be one instance for the metric **kernel.percpu.idle** on a workstation, but multiple instances on a multiprocessor server. Even more dynamic is **filesys.free**, where the values report the amount of free space per file system, and the number of values tracks the mounting and unmounting of local filesystems.

PCP arranges for information describing instance domains to be exported from the performance metric domains to the applications that require this information. Applications may also choose to retrieve values for all instances of a performance metric, or some arbitrary subset of the available instances.

3.2.9 Collector and Monitor Roles

Hosts supporting PCP services are broadly classified into two categories:

1. Collector : Hosts that have **pmcd** and one or more performance metric domain agents (PMDAs) running to collect and export performance metrics
2. Monitor : Hosts that import performance metrics from one or more collector hosts to be consumed by tools to monitor, manage, or record the performance of the collector hosts

Each PCP enabled host can operate as a collector, a monitor, or both.

3.2.10 Retrospective Sources of Performance Metrics

The PMAPI also supports delivery of performance metrics from a historical source in the form of a PCP archive log. Archive logs are created using the **pmlogger** utility, and are replayed in an architecture as shown in *Figure 1.4. Architecture for Retrospective Analysis*.

Fig. 4: Figure 1.4. Architecture for Retrospective Analysis

The PMAPI has been designed to minimize the differences required for an application to process performance data from an archive or from a real-time source. As a result, most PCP tools support live and retrospective monitoring with equal facility.

3.2.11 Product Extensibility

Much of the PCP software's potential for attacking difficult performance problems in production environments comes from the design philosophy that considers extensibility to be critically important.

The performance analyst can take advantage of the PCP infrastructure to deploy value-added performance monitoring tools and services. Here are some examples:

- Easy extension of the PCP collector to accommodate new performance metrics and new sources of performance metrics, in particular using the interfaces of a special-purpose library to develop new PMDAs (see the **pmda(3)** man page)
- Use of libraries (**libpcp_pmda** and **libpcp_mmv**) to aid in the development of new capabilities to export performance metrics from local applications
- Operation on any performance metric using generalized toolkits
- Distribution of PCP components such as collectors across the network, placing the service where it can do the most good
- Dynamic adjustment to changes in system configuration
- Flexible customization built into the design of all PCP tools
- Creation of new monitor applications, using the routines described in the **pmapi(3)** man page

3.3 Overview of Component Software

Performance Co-Pilot (PCP) is composed of both text-based and graphical tools. Each tool is fully documented by a man page. These man pages are named after the tools or commands they describe, and are accessible through the **man** command. For example, to see the **pminfo(1)** man page for the **pminfo** command, enter this command:

```
man pminfo
```

A representative list of PCP tools and commands, grouped by functionality, is provided in the following four sections.

3.3.1 Performance Monitoring and Visualization

The following tools provide the principal services for the PCP end-user with an interest in monitoring, visualizing, or processing performance information collected either in real time or from PCP archive logs:

pcp-atop

Full-screen monitor of the load on a system from a kernel, hardware and processes point of view. It is modeled on the Linux **atop(1)** tool (home page) and provides a showcase for the variety of data available using PCP services and the Python scripting interfaces.

pmchart

Strip chart tool for arbitrary performance metrics. Interactive graphical utility that can display multiple charts simultaneously, from multiple hosts or set of archives, aligned on a unified time axis (X-axis), or on multiple tabs.

pcp-collectl

Statistics collection tool with good coverage of a number of Linux kernel subsystems, with the everything-in-one-tool approach pioneered by **sar(1)**. It is modeled on the Linux **collectl(1)** utility (home page) and provides another example of use of the Python scripting interfaces to build more complex functionality with relative ease, with PCP as a foundation.

pmrep

Outputs the values of arbitrary performance metrics collected live or from a single PCP archive, in textual format.

pmevent

Reports on event metrics, decoding the timestamp and event parameters for text-based reporting.

pmie

Evaluates predicate-action rules over performance metrics for alarms, automated system management tasks, dynamic configuration tuning, and so on. It is an inference engine.

pmieconf

Creates parameterized rules to be used with the PCP inference engine (**pmie**). It can be run either interactively or from scripts for automating the setup of inference (the PCP start scripts do this, for example, to generate a default configuration).

pminfo

Displays information about arbitrary performance metrics available from PCP, including help text with **-T**.

pmlogsummary

Calculates and reports various statistical summaries of the performance metric values from a set of PCP archives.

pmprobe

Probes for performance metric availability, values, and instances.

pmstat

Provides a text-based display of metrics that summarize the performance of one or more systems at a high level.

pmval

Provides a text-based display of the values for arbitrary instances of a selected performance metric, suitable for ASCII logs or inquiry over a slow link.

3.3.2 Collecting, Transporting, and Archiving Performance Information

PCP provides the following tools to support real-time data collection, network transport, and archive log creation services for performance data:

mkaf

Aggregates an arbitrary collection of PCP archive logs into a *folio* to be used with **pmafm**.

pmafm

Interrogates, manages, and replays an archive folio as created by **mkaf**, or the periodic archive log management scripts, or the record mode of other PCP tools.

pmcd

Is the Performance Metrics Collection Daemon (PMCD). This daemon must run on each system being monitored, to collect and export the performance information necessary to monitor the system.

pmcd_wait

Waits for **pmcd** to be ready to accept client connections.

pmdaapache

Exports performance metrics from the Apache Web Server. It is a Performance Metrics Domain Agent (PMDA).

pmdacisco

Extracts performance metrics from one or more Cisco routers.

pmdaelasticseach

Extracts performance metrics from an elasticsearch cluster.

pmdagfs2

Exports performance metrics from the GFS2 clustered filesystem.

pmdagluster

Extracts performance metrics from the Gluster filesystem.

pmdainfiniband

Exports performance metrics from the Infiniband kernel driver.

pmdakvm

Extracts performance metrics from the Linux Kernel Virtual Machine (KVM) infrastructure.

pmdalustrecomm

Exports performance metrics from the Lustre clustered filesystem.

pmdamailq

Exports performance metrics describing the current state of items in the **sendmail** queue.

pmdamemcache

Extracts performance metrics from memcached, a distributed memory caching daemon commonly used to improve web serving performance.

pmdammv

Exports metrics from instrumented applications linked with the **pcp_mmv** shared library or the Parfait framework for Java instrumentation. These metrics are custom developed per application, and in the case of Parfait, automatically include numerous JVM, Tomcat and other server or container statistics.

pmdamysql

Extracts performance metrics from the MySQL relational database.

pmdanamed

Exports performance metrics from the Internet domain name server, named.

pmdanginx

Extracts performance metrics from the nginx HTTP and reverse proxy server.

pmdapostfix

Export performance metrics from the Postfix mail transfer agent.

pmdapostgres

Extracts performance metrics from the PostgreSQL relational database.

pmdaproc

Exports performance metrics for running processes.

pmdarsyslog

Extracts performance metrics from the Reliable System Log daemon.

pmdasamba

Extracts performance metrics from Samba, a Windows SMB/CIFS server.

pmdasendmail

Exports mail activity statistics from **sendmail**.

pmdashping

Exports performance metrics for the availability and quality of service (response-time) for arbitrary shell commands.

pmdasnmp

Extracts SNMP performance metrics from local or remote SNMP-enabled devices.

pmdasummary

Derives performance metrics values from values made available by other PMDAs. It is a PMDA itself.

pmdasystemd

Extracts performance metrics from the systemd and journald services.

pmdatrace

Exports transaction performance metrics from application processes that use the **pcp_trace** library.

pmdavmware

Extracts performance metrics from a VMWare virtualization host.

pmdaweblog

Scans Web-server logs to extract metrics characterizing.

pmdaxfs

Extracts performance metrics from the Linux kernel XFS filesystem implementation.

pmdumplog

Displays selected state information, control data, and metric values from a set of PCP archive logs created by **pmlogger**.

pmlc

Exercises control over an instance of the PCP archive logger **pmlogger**, to modify the profile of which metrics are logged and/or how frequently their values are logged.

pmlogcheck

Performs integrity check for individual PCP archives.

pmlogconf

Creates or modifies **pmlogger** configuration files for many common logging scenarios, optionally probing for available metrics and enabled functionality. It can be run either interactively or from scripts for automating the setup of data logging (the PCP start scripts do this, for example, to generate a default configuration).

pmlogextract

Reads one or more PCP archive logs and creates a temporally merged and reduced PCP archive log as output.

pmlogger

Creates PCP archive logs of performance metrics over time. Many tools accept these PCP archive logs as alternative sources of metrics for retrospective analysis.

pmproxy

Provides REST APIs, archive discovery, and both PCP and Redis protocol proxying when executing PCP or Redis client tools through a network firewall system.

pmtrace

Provides a simple command line interface to the trace PMDA and its associated **pcp_trace** library.

3.3.3 Operational and Infrastructure Support

PCP provides the following tools to support the PCP infrastructure and assist operational procedures for PCP deployment in a production environment:

pcp

Summarizes that state of a PCP installation.

pmdbg

Describes the available facilities and associated control flags. PCP tools include internal diagnostic and debugging facilities that may be activated by run-time flags.

pmerr

Translates PCP error codes into human-readable error messages.

pmhostname

Reports hostname as returned by **gethostbyname**. Used in assorted PCP management scripts.

pmie_check

Administration of the Performance Co-Pilot inference engine (**pmie**).

pmlock

Attempts to acquire an exclusive lock by creating a file with a mode of 0.

pmlogger_*

Allows you to create a customized regime of administration and management for PCP archive log files. The **pmlogger_check**, **pmlogger_daily**, and **pmlogger_merge** scripts are intended for periodic execution via the **cron** command.

pmnewlog

Performs archive log rotation by stopping and restarting an instance of **pmlogger**.

pmnsadd

Adds a subtree of new names into a PMNS, as used by the components of PCP.

pmnsdel

Removes a subtree of names from a PMNS, as used by the components of the PCP.

pmnsmerge

Merges multiple PMNS files together, as used by the components of PCP.

pmstore

Reinitializes counters or assigns new values to metrics that act as control variables. The command changes the current values for the specified instances of a single performance metric.

3.3.4 Application and Agent Development

The following PCP tools aid the development of new programs to consume performance data, and new agents to export performance data within the PCP framework:

chkhelp

Checks the consistency of performance metrics help database files.

dbpmda

Allows PMDA behavior to be exercised and tested. It is an interactive debugger for PMDAs.

newhelp

Generates the database files for one or more source files of PCP help text.

pmapi

Defines a procedural interface for developing PCP client applications. It is the Performance Metrics Application Programming Interface (PMAPI).

pmclient

Is a simple client that uses the PMAPI to report some high-level system performance metrics.

pmda

Is a library used by many shipped PMDAs to communicate with a **pmcd** process. It can expedite the development of new and custom PMDAs.

pmgenmap

Generates C declarations and **cpp(1)** macros to aid the development of customized programs that use the facilities of PCP. It is a PMDA development tool.

CHAPTER

FOUR

INSTALLING AND CONFIGURING PERFORMANCE CO-PILOT

Contents

- *Installing and Configuring Performance Co-Pilot*
  - *Product Structure*
  - *Performance Metrics Collection Daemon (PMCD)*
    - *Starting and Stopping the PMCD*
    - *Restarting an Unresponsive PMCD*
    - *PMCD Diagnostics and Error Messages*
    - *PMCD Options and Configuration Files*
      - *The pmcd.options File*
      - *The pmcd.conf File*
      - *Controlling Access to PMCD with pmcd.conf*
  - *Managing Optional PMDAs*
    - *PMDA Installation on a PCP Collector Host*
    - *PMDA Removal on a PCP Collector Host*
  - *Troubleshooting*
    - *Performance Metrics Name Space*
    - *Missing and Incomplete Values for Performance Metrics*
      - *Metric Values Not Available*
    - *Kernel Metrics and the PMCD*
      - *Cannot Connect to Remote PMCD*
      - *PMCD Not Reconfiguring after SIGHUP*
      - *PMCD Does Not Start*

The sections in this chapter describe the basic installation and configuration steps necessary to run Performance Co-Pilot (PCP) on your systems. The following major sections are included:

Section 2.1, "*Product Structure*" describes the main packages of PCP software and how they must be installed on each system.

Section 2.2, "*Performance Metrics Collection Daemon (PMCD)*", describes the fundamentals of maintaining the performance data collector.

Section 2.3, "*Managing Optional PMDAs*", describes the basics of installing a new Performance Metrics Domain Agent (PMDA) to collect metric data and pass it to the PMCD.

Section 2.4, "*Troubleshooting*", offers advice on problems involving the PMCD.

4.1 Product Structure

In a typical deployment, Performance Co-Pilot (PCP) would be installed in a collector configuration on one or more hosts, from which the performance information could then be collected, and in a monitor configuration on one or more workstations, from which the performance of the server systems could then be monitored.

On some platforms Performance Co-Pilot is presented as multiple packages; typically separating the server components from graphical user interfaces and documentation.

pcp-X.Y.Z-rev	package for core PCP
pcp-gui-X.Y.Z-rev	package for graphical PCP client tools
pcp-doc-X.Y.Z-rev	package for online PCP documentation

4.2 Performance Metrics Collection Daemon (PMCD)

On each Performance Co-Pilot (PCP) collection system, you must be certain that the **pmcd** daemon is running. This daemon coordinates the gathering and exporting of performance statistics in response to requests from the PCP monitoring tools.

4.2.1 Starting and Stopping the PMCD

To start the daemon, enter the following commands as root on each PCP collection system:

```
chkconfig pmcd on
${PCP_RC_DIR}/pmcd start
```

These commands instruct the system to start the daemon immediately, and again whenever the system is booted. It is not necessary to start the daemon on the monitoring system unless you wish to collect performance information from it as well.

To stop **pmcd** immediately on a PCP collection system, enter the following command:

```
${PCP_RC_DIR}/pmcd stop
```

4.2.2 Restarting an Unresponsive PMCD

Sometimes, if a daemon is not responding on a PCP collection system, the problem can be resolved by stopping and then immediately restarting a fresh instance of the daemon. If you need to stop and then immediately restart PMCD on a PCP collection system, use the **start** argument provided with the script in `${PCP_RC_DIR}`. The command syntax is, as follows:

```
${PCP_RC_DIR}/pmcd start
```

On startup, **pmcd** looks for a configuration file at `${PCP_PMCDCONF_PATH}`. This file specifies which agents cover which performance metrics domains and how PMCD should make contact with the agents. A comprehensive description of the configuration file syntax and semantics can be found in the **pmcd(1)** man page.

If the configuration is changed, **pmcd** reconfigures itself when it receives the **SIGHUP** signal. Use the following command to send the **SIGHUP** signal to the daemon:

```
${PCP_BINADM_DIR}/pmsignal -a -s HUP pmcd
```

This is also useful when one of the PMDAs managed by **pmcd** has failed or has been terminated by **pmcd**. Upon receipt of the **SIGHUP** signal, **pmcd** restarts any PMDA that is configured but inactive. The exception to this rule is the case of a PMDA which must run with superuser privileges (where possible, this is avoided) - for these PMDAs, a full **pmcd** restart must be performed, using the process described earlier (not SIGHUP).

4.2.3 PMCD Diagnostics and Error Messages

If there is a problem with **pmcd**, the first place to investigate should be the **pmcd.log** file. By default, this file is in the `${PCP_LOG_DIR}/pmcd` directory.

4.2.4 PMCD Options and Configuration Files

There are two files that control PMCD operation. These are the `${PCP_PMCDCONF_PATH}` and `${PCP_PMCDOPTIONS_PATH}` files. The **pmcd.options** file contains the command line options used with PMCD; it is read when the daemon is invoked by `${PCP_RC_DIR}/pmcd`. The **pmcd.conf** file contains configuration information regarding domain agents and the metrics that they monitor. These configuration files are described in the following sections.

The pmcd.options File

Command line options for the PMCD are stored in the `${PCP_PMCDOPTIONS_PATH}` file. The PMCD can be invoked directly from a shell prompt, or it can be invoked by `${PCP_RC_DIR}/pmcd` as part of the boot process. It is usual and normal to invoke it using `${PCP_RC_DIR}/pmcd`, reserving shell invocation for debugging purposes.

The PMCD accepts certain command line options to control its execution, and these options are placed in the **pmcd.options** file when `${PCP_RC_DIR}/pmcd` is being used to start the daemon. The following options (amongst others) are available:

-i *address*

For hosts with more than one network interface, this option specifies the interface on which this instance of the PMCD accepts connections. Multiple **-i** options may be specified. The default in the absence of any **-i** option is for PMCD to accept connections on all interfaces.

-l *file*

Specifies a log file. If no **-l** option is specified, the log file name is **pmcd.log** and it is created in the directory `${PCP_LOG_DIR}/pmcd/`.

-s *file*

Specifies the path to a local unix domain socket (for platforms supporting this socket family only). The default value is `${PCP_RUN_DIR}/pmcd.socket`.

-t *seconds*

Specifies the amount of time, in seconds, before PMCD times out on protocol data unit (PDU) exchanges with PMDAs. If no time out is specified, the default is five seconds. Setting time out to zero disables time outs (not recommended, PMDAs should always respond quickly).

The time out may be dynamically modified by storing the number of seconds into the metric pmcd.control.timeout using pmstore.

-T *mask*

Specifies whether connection and PDU tracing are turned on for debugging purposes.

See the **pmcd(1)** man page for complete information on these options.

The default **pmcd.options** file shipped with PCP is similar to the following:

```
# command-line options to pmcd, uncomment/edit lines as required

# longer timeout delay for slow agents
# -t 10

# suppress timeouts
# -t 0

# make log go someplace else
# -l /some/place/else

# debugging knobs, see pmdbg(1)
# -D N
# -f

# Restricting (further) incoming PDU size to prevent DOS attacks
# -L 16384

# enable event tracing bit fields
#   1   trace connections
#   2   trace PDUs
# 256   unbuffered tracing
# -T 3

# setting of environment variables for pmcd and
# the PCP rc scripts. See pmcd(1) and PMAPI(3).
# PMCD_WAIT_TIMEOUT=120
```

The most commonly used options have been placed in this file for your convenience. To uncomment and use an option, simply remove the pound sign (#) at the beginning of the line with the option you wish to use. Restart **pmcd** for the change to take effect; that is, as superuser, enter the command:

```
${PCP_RC_DIR}/pmcd start
```

The pmcd.conf File

When the PMCD is invoked, it reads its configuration file, which is `${PCP_PMCDCONF_PATH}`. This file contains entries that specify the PMDAs used by this instance of the PMCD and which metrics are covered by these PMDAs. Also, you may specify access control rules in this file for the various hosts, users and groups on your network. This file is described completely in the **pmcd(1)** man page.

With standard PCP operation (even if you have not created and added your own PMDAs), you might need to edit this file in order to add any additional access control you wish to impose. If you do not add access control rules, all access for all operations is granted to the local host, and read-only access is granted to remote hosts. The **pmcd.conf** file is automatically generated during the software build process and on Linux, for example, is similar to the following:

```
 Performance Metrics Domain Specifications
#
# This file is automatically generated during the build
# Name  Id      IPC     IPC Params      File/Cmd
root    1       pipe    binary          /var/lib/pcp/pmdas/root/pmdaroot
pmcd    2       dso     pmcd_init       ${PCP_PMDAS_DIR}/pmcd/pmda_pmcd.so
proc    3       pipe    binary          ${PCP_PMDAS_DIR}/proc/pmdaproc -d 3
xfs     11      pipe    binary          ${PCP_PMDAS_DIR}/xfs/pmdaxfs -d 11
linux   60      dso     linux_init      ${PCP_PMDAS_DIR}/linux/pmda_linux.so
mmv     70      dso     mmv_init        /var/lib/pcp/pmdas/mmv/pmda_mmv.so

[access]
disallow ".*" : store;
disallow ":*" : store;
allow "local:*" : all;
```

Note: Even though PMCD does not run with **root** privileges, you must be very careful not to configure PMDAs in this file if you are not sure of their action. This is because all PMDAs are initially started as **root** (allowing them to assume alternate identities, such as **postgres** for example), after which **pmcd** drops its privileges. Pay close attention that permissions on this file are not inadvertently downgraded to allow public write access.

Each entry in this configuration file contains rules that specify how to connect the PMCD to a particular PMDA and which metrics the PMDA monitors. A PMDA may be attached as a Dynamic Shared Object (DSO) or by using a socket or a pair of pipes. The distinction between these attachment methods is described below.

An entry in the **pmcd.conf** file looks like this:

```
label_name   domain_number   type   path
```

The *label_name* field specifies a name for the PMDA. The *domain_number* is an integer value that specifies a domain of metrics for the PMDA. The *type* field indicates the type of entry (DSO, socket, or pipe). The *path* field is for additional information, and varies according to the type of entry.

The following rules are common to DSO, socket, and pipe syntax:

label_name	An alphanumeric string identifying the agent.
domain_number	An unsigned integer specifying the agent's domain.

DSO entries follow this syntax:

```
label_name domain_number dso entry-point path
```

The following rules apply to the DSO syntax:

dso	The entry type.
entry-point	The name of an initialization function called when the DSO is loaded.
path	Designates the location of the DSO. An absolute path must be used. On most platforms this will be a **so** suffixed file, on Windows it is a **dll**, and on Mac OS X it is a **dylib** file.

Socket entries in the **pmcd.conf** file follow this syntax:

```
label_name domain_number socket addr_family address command [args]
```

The following rules apply to the socket syntax:

socket	The entry type.
addr_family	Specifies if the socket is **AF_INET**, **AF_IPV6** or **AF_UNIX**. If the socket is **INET**, the word **inet** appears in this place. If the socket is **IPV6**, the word **ipv6** appears in this place. If the socket is **UNIX**, the word **unix** appears in this place.
address	Specifies the address of the socket. For INET or IPv6 sockets, this is a port number or port name. For UNIX sockets, this is the name of the PMDA's socket on the local host.
command	Specifies a command to start the PMDA when the PMCD is invoked and reads the configuration file.
args	Optional arguments for *command*.

Pipe entries in the **pmcd.conf** file follow this syntax:

```
label_name domain_number pipe protocol command [args]
```

The following rules apply to the pipe syntax:

pipe	The entry type.
protocol	Specifies whether a text-based or a binary PCP protocol should be used over the pipes. Historically, this parameter was able to be "text" or "binary." The text-based protocol has long since been deprecated and removed, however, so nowadays "binary" is the only valid value here.
command	Specifies a command to start the PMDA when the PMCD is invoked and reads the configuration file.
args	Optional arguments for command.

Controlling Access to PMCD with pmcd.conf

You can place this option extension in the **pmcd.conf** file to control access to performance metric data based on hosts, users and groups. To add an access control section, begin by placing the following line at the end of your **pmcd.conf** file:

```
[access]
```

Below this line, you can add entries of the following forms:

```
allow hosts hostlist : operations ;   disallow hosts hostlist : operations ;
allow users userlist : operations ;   disallow users userlist : operations ;
allow groups grouplist : operations ;   disallow groups grouplist : operations ;
```

The keywords *users*, *groups* and *hosts* can be used in either plural or singular form.

The *userlist* and *grouplist* fields are comma-separated lists of authenticated users and groups from the local **/etc/passwd** and **/etc/groups** files, NIS (network information service) or LDAP (lightweight directory access protocol) service.

The *hostlist* is a comma-separated list of host identifiers; the following rules apply:

- Host names must be in the local system's **/etc/hosts** file or known to the local DNS (domain name service).
- IP and IPv6 addresses may be given in the usual numeric notations.
- A wildcarded IP or IPv6 address may be used to specify groups of hosts, with the single wildcard character * as the last-given component of the address. The wildcard .* refers to all IP (IPv4) addresses. The wildcard :* refers to all IPv6 addresses. If an IPv6 wildcard contains a :: component, then the final * refers to the final 16 bits of the address only, otherwise it refers to the remaining unspecified bits of the address.

The wildcard "*" refers to all users, groups or host addresses. Names of users, groups or hosts may not be wildcarded.

For example, the following *hostlist* entries are all valid:

```
babylon
babylon.acme.com
123.101.27.44
localhost
155.116.24.*
192.*
.*
fe80::223:14ff:feaf:b62c
fe80::223:14ff:feaf:*
fe80:*
:*
*
```

The operations field can be any of the following:

- A comma-separated list of the operation types described below.
- The word all to allow or disallow all operations as specified in the first field.
- The words all except and a list of operations. This entry allows or disallows all operations as specified in the first field except those listed.
- The phrase maximum N connections to set an upper bound (N) on the number of connections an individual host, user or group of users may make. This can only be added to the operations list of an allow statement.

The operations that can be allowed or disallowed are as follows:

- fetch : Allows retrieval of information from the PMCD. This may be information about a metric (such as a description, instance domain, or help text) or an actual value for a metric.
- store : Allows the PMCD to store metric values in PMDAs that permit store operations. Be cautious in allowing this operation, because it may be a security opening in large networks, although the PMDAs shipped with the PCP package typically reject store operations, except for selected performance metrics where the effect is benign.

For example, here is a sample access control portion of a `${PCP_PMCDCONF_PATH}` file:

```
allow hosts babylon, moomba : all ;
disallow user sam : all ;
allow group dev : fetch ;
allow hosts 192.127.4.* : fetch ;
disallow host gate-inet : store ;
```

Complete information on access control syntax rules in the **pmcd.conf** file can be found in the **pmcd(1)** man page.

4.3 Managing Optional PMDAs

Some Performance Metrics Domain Agents (PMDAs) shipped with Performance Co-Pilot (PCP) are designed to be installed and activated on every collector host, for example, **linux**, **windows**, **darwin**, **pmcd**, and **process** PMDAs.

Other PMDAs are designed for optional activation and require some user action to make them operational. In some cases these PMDAs expect local site customization to reflect the operational environment, the system configuration, or the production workload. This customization is typically supported by interactive installation scripts for each PMDA.

Each PMDA has its own directory located below `${PCP_PMDAS_DIR}`. Each directory contains a **Remove** script to unconfigure the PMDA, remove the associated metrics from the PMNS, and restart the **pmcd** daemon; and an **Install** script to install the PMDA, update the PMNS, and restart the PMCD daemon.

As a shortcut mechanism to support automated PMDA installation, a file named **.NeedInstall** can be created in a PMDA directory below `${PCP_PMDAS_DIR}`. The next restart of PCP services will invoke that PMDAs installation automatically, with default options taken.

4.3.1 PMDA Installation on a PCP Collector Host

To install a PMDA you must perform a collector installation for each host on which the PMDA is required to export performance metrics. PCP provides a distributed metric namespace (PMNS) and metadata, so it is not necessary to install PMDAs (with their associated PMNS) on PCP monitor hosts.

You need to update the PMNS, configure the PMDA, and notify PMCD. The **Install** script for each PMDA automates these operations, as follows:

1. Log in as **root** (the superuser).
2. Change to the PMDA's directory as shown in the following example:

```
cd ${PCP_PMDAS_DIR}/cisco
```

3. In the unlikely event that you wish to use a non-default Performance Metrics Domain (PMD) assignment, determine the current PMD assignment:

```
cat domain.h
```

Check that there is no conflict in the PMDs as defined in `${PCP_VAR_DIR}/pmns/stdpmid` and the other PMDAs currently in use (listed in `${PCP_PMCDCONF_PATH}`). Edit **domain.h** to assign the new domain number if there is a conflict (this is highly unlikely to occur in a regular PCP installation).

4. Enter the following command:

```
./Install
```

You may be prompted to enter some local parameters or configuration options. The script applies all required changes to the control files and to the PMNS, and then notifies PMCD. Example 2.1, "PMNS Installation Output " is illustrative of the interactions:

```
Example 2.1. PMNS Installation Output

Cisco hostname or IP address? [return to quit] wanmelb

A user-level password may be required for Cisco "show int" command.
    If you are unsure, try the command
        $ telnet wanmelb
    and if the prompt "Password:" appears, a user-level password is
```

(continues on next page)

(continued from previous page)

```
    required; otherwise answer the next question with an empty line.

User-level Cisco password? ********
Probing Cisco for list of interfaces ...

Enter interfaces to monitor, one per line in the format
tX where "t" is a type and one of "e" (Ethernet), or "f" (Fddi), or
"s" (Serial), or "a" (ATM), and "X" is an interface identifier
which is either an integer (e.g.  4000 Series routers) or two
integers separated by a slash (e.g. 7000 Series routers).

The currently unselected interfaces for the Cisco "wanmelb" are:
    e0 s0 s1
Enter "quit" to terminate the interface selection process.
    Interface? [e0] s0

The currently unselected interfaces for the Cisco "wanmelb" are:
    e0 s1
Enter "quit" to terminate the interface selection process.
    Interface? [e0] **s1**

The currently unselected interfaces for the Cisco "wanmelb" are:
    e0
Enter "quit" to terminate the interface selection process.
Interface? [e0] **quit**

Cisco hostname or IP address? [return to quit]
Updating the Performance Metrics Name Space (PMNS) ...
Installing pmchart view(s) ...
Terminate PMDA if already installed ...
Installing files ...
Updating the PMCD control file, and notifying PMCD ...
Check cisco metrics have appeared ... 5 metrics and 10 values
```

4.3.2 PMDA Removal on a PCP Collector Host

To remove a PMDA, you must perform a collector removal for each host on which the PMDA is currently installed.

The PMNS needs to be updated, the PMDA unconfigured, and PMCD notified. The **Remove** script for each PMDA automates these operations, as follows:

1. Log in as **root** (the superuser).
2. Change to the PMDA's directory as shown in the following example:

   ```
   cd ${PCP_PMDAS_DIR}/elasticsearch
   ```

3. Enter the following command:

   ```
   ./Remove
   ```

The following output illustrates the result:

```
    Culling the Performance Metrics Name Space ...
   elasticsearch ... done
Updating the PMCD control file, and notifying PMCD ...
```

(continues on next page)

(continued from previous page)

```
Removing files ...
Check elasticsearch metrics have gone away ... OK
```

4.4 Troubleshooting

The following sections offer troubleshooting advice on the Performance Metrics Name Space (PMNS), missing and incomplete values for performance metrics, kernel metrics and the PMCD.

Advice for troubleshooting the archive logging system is provided in Chapter 6, Archive Logging.

4.4.1 Performance Metrics Name Space

To display the active PMNS, use the **pminfo** command; see the **pminfo(1)** man page.

The PMNS at the collector host is updated whenever a PMDA is installed or removed, and may also be updated when new versions of PCP are installed. During these operations, the PMNS is typically updated by merging the (plaintext) namespace components from each installed PMDA. These separate PMNS components reside in the `${PCP_VAR_DIR}/pmns` directory and are merged into the **root** file there.

4.4.2 Missing and Incomplete Values for Performance Metrics

Missing or incomplete performance metric values are the result of their unavailability.

Metric Values Not Available

The following symptom has a known cause and resolution:

Symptom:

Values for some or all of the instances of a performance metric are not available.

Cause:

This can occur as a consequence of changes in the installation of modules (for example, a DBMS or an application package) that provide the performance instrumentation underpinning the PMDAs. Changes in the selection of modules that are installed or operational, along with changes in the version of these modules, may make metrics appear and disappear over time.

In simple terms, the PMNS contains a metric name, but when that metric is requested, no PMDA at the collector host supports the metric.

For archive logs, the collection of metrics to be logged is a subset of the metrics available, so utilities replaying from a PCP archive log may not have access to all of the metrics available from a live (PMCD) source.

Resolution:

Make sure the underlying instrumentation is available and the module is active. Ensure that the PMDA is running on the host to be monitored. If necessary, create a new archive log with a wider range of metrics to be logged.

4.4.3 Kernel Metrics and the PMCD

The following issues involve the kernel metrics and the PMCD:

- Cannot connect to remote PMCD
- PMCD not reconfiguring after hang-up
- PMCD does not start

Cannot Connect to Remote PMCD

The following symptom has a known cause and resolution:

Symptom:

A PCP client tool (such as **pmchart**, **pmie**, or **pmlogger**) complains that it is unable to connect to a remote PMCD (or establish a PMAPI context), but you are sure that PMCD is active on the remote host.

Cause:

To avoid hanging applications for the duration of TCP/IP time outs, the PMAPI library implements its own time out when trying to establish a connection to a PMCD. If the connection to the host is over a slow network, then successful establishment of the connection may not be possible before the time out, and the attempt is abandoned.

Alternatively, there may be a firewall in-between the client tool and PMCD which is blocking the connection attempt.

Finally, PMCD may be running in a mode where it does not accept remote connections, or only listening on certain interface.

Resolution:

Establish that the PMCD on far-away-host is really alive, by connecting to its control port (TCP port number 44321 by default):

```
telnet far-away-host 44321
```

This response indicates the PMCD is not running and needs restarting:

```
Unable to connect to remote host: Connection refused
```

To restart the PMCD on that host, enter the following command:

```
${PCP_RC_DIR}/pmcd start
```

This response indicates the PMCD is running:

```
Connected to far-away-host
```

Interrupt the **telnet** session, increase the PMAPI time out by setting the **PMCD_CONNECT_TIMEOUT** environment variable to some number of seconds (60 for instance), and try the PCP client tool again.

Verify that PMCD is not running in local-only mode, by looking for an enabled value (one) from:

```
pminfo -f pmcd.feature.local
```

This setting is controlled from the **PMCD_LOCAL** environment variable usually set via `${PCP_SYSCONFIG_DIR}/pmcd`.

If these techniques are ineffective, it is likely an intermediary firewall is blocking the client from accessing the PMCD port - resolving such issues is firewall-host platform-specific and cannot practically be covered here.

PMCD Not Reconfiguring after SIGHUP

The following symptom has a known cause and resolution:

Symptom:

PMCD does not reconfigure itself after receiving the **SIGHUP** signal.

Cause:

If there is a syntax error in `${PCP_PMCDCONF_PATH}`, PMCD does not use the contents of the file. This can lead to situations in which the configuration file and PMCD's internal state do not agree.

Resolution:

Always monitor PMCD's log. For example, use the following command in another window when reconfiguring PMCD, to watch errors occur:

```
tail -f ${PCP_LOG_DIR}/pmcd/pmcd.log
```

PMCD Does Not Start

The following symptom has a known cause and resolution:

Symptom:

If the following messages appear in the PMCD log (`${PCP_LOG_DIR}/pmcd/pmcd.log`), consider the cause and resolution:

```
pcp[27020] Error: OpenRequestSocket(44321) bind: Address already in
use
pcp[27020] Error: pmcd is already running
pcp[27020] Error: pmcd not started due to errors!
```

Cause:

PMCD is already running or was terminated before it could clean up properly. The error occurs because the socket it advertises for client connections is already being used or has not been cleared by the kernel.

Resolution:

Start PMCD as **root** (superuser) by typing:

```
${PCP_RC_DIR}/pmcd start
```

Any existing PMCD is shut down, and a new one is started in such a way that the symptomatic message should not appear.

If you are starting PMCD this way and the symptomatic message appears, a problem has occurred with the connection to one of the deceased PMCD's clients.

This could happen when the network connection to a remote client is lost and PMCD is subsequently terminated. The system may attempt to keep the socket open for a time to allow the remote client a chance to reestablish the connection and read any outstanding data.

The only solution in these circumstances is to wait until the socket times out and the kernel deletes it. This netstat command displays the status of the socket and any connections:

```
netstat -ant | grep 44321
```

If the socket is in the **FIN_WAIT** or **TIME_WAIT** state, then you must wait for it to be deleted. Once the command above produces no output, PMCD may be restarted. Less commonly, you may have another program running on your system that uses the same Internet port number (44321) that PMCD uses.

Refer to the **PCPIntro(1)** man page for a description of how to override the default PMCD port assignment using the **PMCD_PORT** environment variable.

CHAPTER

FIVE

COMMON CONVENTIONS AND ARGUMENTS

Contents

- *Common Conventions and Arguments*
  - *Alternate Metrics Source Options*
    - *Fetching Metrics from Another Host*
    - *Fetching Metrics from an Archive Log*
  - *General PCP Tool Options*
    - *Common Directories and File Locations*
    - *Alternate Performance Metric Name Spaces*
  - *Time Duration and Control*
    - *Performance Monitor Reporting Frequency and Duration*
    - *Time Window Options*
    - *Timezone Options*
  - *PCP Environment Variables*
  - *Running PCP Tools through a Firewall*
    - *The pmproxy service*
  - *Transient Problems with Performance Metric Values*
    - *Performance Metric Wraparound*
    - *Time Dilation and Time Skew*

This chapter deals with the user interface components that are common to most text-based utilities that make up the monitor portion of Performance Co-Pilot (PCP). These are the major sections in this chapter:

Section 3.1, "*Alternate Metrics Source Options*", details some basic standards used in the development of PCP tools.

Section 3.2, "*General PCP Tool Options*", details other options to use with PCP tools.

Section 3.3, "*Time Duration and Control*", describes the time control dialog and time-related command line options available for use with PCP tools.

Section 3.4, "*PCP Environment Variables*", describes the environment variables supported by PCP tools.

Section 3.5, "*Running PCP Tools through a Firewall*", describes how to execute PCP tools that must retrieve performance data from the Performance Metrics Collection Daemon (PMCD) on the other side of a TCP/IP security

firewall.

Section 3.6, "*Transient Problems with Performance Metric Values*", covers some uncommon scenarios that may compromise performance metric integrity over the short term.

Many of the utilities provided with PCP conform to a common set of naming and syntactic conventions for command line arguments and options. This section outlines these conventions and their meaning. The options may be generally assumed to be honored for all utilities supporting the corresponding functionality.

In all cases, the man pages for each utility fully describe the supported command arguments and options.

Command line options are also relevant when starting PCP applications from the desktop using the **Alt** double-click method. This technique launches the **pmrun** program to collect additional arguments to pass along when starting a PCP application.

5.1 Alternate Metrics Source Options

The default source of performance metrics is from PMCD on the local host. This default **pmcd** connection will be made using the Unix domain socket, if the platform supports that, else a localhost Inet socket connection is made. This section describes how to obtain metrics from sources other than this default.

5.1.1 Fetching Metrics from Another Host

The option **-h** *host* directs any PCP utility (such as **pmchart** or **pmie**) to make a connection with the PMCD instance running on *host*. Once established, this connection serves as the principal real-time source of performance metrics and metadata. The *host* specification may be more than a simple host name or address - it can also contain decorations specifying protocol type (secure or not), authentication information, and other connection attributes. Refer to the **PCPIntro(1)** man page for full details of these, and examples of use of these specifications can also be found in the *PCP Tutorials and Case Studies* companion document.

5.1.2 Fetching Metrics from an Archive Log

The option **-a** *archive* directs the utility to treat the set of PCP archive logs designated by archive as the principal source of performance metrics and metadata. archive is a comma-separated list of names, each of which may be the base name of an archive or the name of a directory containing archives.

PCP archive logs are created with **pmlogger**. Most PCP utilities operate with equal facility for performance information coming from either a real-time feed via PMCD on some host, or for historical data from a set of PCP archive logs. For more information on archive logs and their use, see Chapter 6, Archive Logging.

The list of names (**archive**) used with the **-a** option implies the existence of the files created automatically by **pmlogger**, as listed in Table 3.1, "Physical Filenames for Components of a PCP Archive Log".

Table 3.1. Physical Filenames for Components of a PCP Archive Log

Filename	Contents
archive. *index*	Temporal index for rapid access to archive contents
archive. *meta*	Metadata descriptions for performance metrics and instance domains appearing in the archive
archive.N	Volumes of performance metrics values, for **N** = 0,1,2,...

Most tools are able to concurrently process multiple PCP archive logs (for example, for retrospective analysis of performance across multiple hosts), and accept either multiple **-a** options or a comma separated list of archive names following the **-a** option.

Note: The **-h** and **-a** options are almost always mutually exclusive. Currently, **pmchart** is the exception to this rule but other tools may continue to blur this line in the future.

5.2 General PCP Tool Options

The following sections provide information relevant to most of the PCP tools. It is presented here in a single place for convenience.

5.2.1 Common Directories and File Locations

The following files and directories are used by the PCP tools as repositories for option and configuration files and for binaries:

`${PCP_DIR}/etc/pcp.env`

Script to set PCP run-time environment variables.

`${PCP_DIR}/etc/pcp.conf`

PCP configuration and environment file.

`${PCP_PMCDCONF_PATH}`

Configuration file for Performance Metrics Collection Daemon (PMCD). Sets environment variables, including **PATH**.

`${PCP_BINADM_DIR}/pmcd`

The PMCD binary.

`${PCP_PMCDOPTIONS_PATH}`

Command line options for PMCD.

`${PCP_RC_DIR}/pmcd`

The PMCD startup script.

`${PCP_BIN_DIR}/pcptool`

Directory containing PCP tools such as **pmstat , pminfo, pmlogger, pmlogsummary, pmchart, pmie,** and so on.

`${PCP_SHARE_DIR}`

Directory containing shareable PCP-specific files and repository directories such as **bin, demos, examples** and **lib**.

`${PCP_VAR_DIR}`

Directory containing non-shareable (that is, per-host) PCP specific files and repository directories.

`${PCP_BINADM_DIR}/pcptool`

PCP tools that are typically not executed directly by the end user such as **pmcd_wait**.

`${PCP_SHARE_DIR}/lib/pcplib`

Miscellaneous PCP libraries and executables.

`${PCP_PMDAS_DIR}`

Performance Metric Domain Agents (PMDAs), one directory per PMDA.

`${PCP_VAR_DIR}/config`

Configuration files for PCP tools, typically with one directory per tool.

`${PCP_DEMOS_DIR}`

Demonstration data files and example programs.

`${PCP_LOG_DIR}`

By default, diagnostic and trace log files generated by PMCD and PMDAs. Also, the PCP archive logs are managed in one directory per logged host below here.

`${PCP_VAR_DIR}/pmns`

Files and scripts for the Performance Metrics Name Space (PMNS).

5.2.2 Alternate Performance Metric Name Spaces

The Performance Metrics Name Space (PMNS) defines a mapping from a collection of human-readable names for performance metrics (convenient to the user) into corresponding internal identifiers (convenient for the underlying implementation).

The distributed PMNS used in PCP avoids most requirements for an alternate PMNS, because clients' PMNS operations are supported at the Performance Metrics Collection Daemon (PMCD) or by means of PMNS data in a PCP archive log. The distributed PMNS is the default, but alternates may be specified using the **-n** *namespace* argument to the PCP tools. When a PMNS is maintained on a host, it is likely to reside in the `${PCP_VAR_DIR}/pmns` directory.

5.3 Time Duration and Control

The periodic nature of sampling performance metrics and refreshing the displays of the PCP tools makes specification and control of the temporal domain a common operation. In the following sections, the services and conventions for specifying time positions and intervals are described.

5.3.1 Performance Monitor Reporting Frequency and Duration

Many of the performance monitoring utilities have periodic reporting patterns. The **-t** *interval* and **-s** *samples* options are used to control the sampling (reporting) interval, usually expressed as a real number of seconds (*interval*), and the number of samples to be reported, respectively. In the absence of the **-s** flag, the default behavior is for the performance monitoring utilities to run until they are explicitly stopped.

The *interval* argument may also be expressed in terms of minutes, hours, or days, as described in the **PCPIntro(1)** man page.

5.3.2 Time Window Options

The following options may be used with most PCP tools (typically when the source of the performance metrics is a PCP archive log) to tailor the beginning and end points of a display, the sample origin, and the sample time alignment to your convenience.

The **-S, -T, -O** and **-A** command line options are used by PCP applications to define a time window of interest.

-S *duration*

The start option may be used to request that the display start at the nominated time. By default, the first sample of performance data is retrieved immediately in real-time mode, or coincides with the first sample of data of the first

archive in a set of PCP archive logs in archive mode. For archive mode, the **-S** option may be used to specify a later time for the start of sampling. By default, if duration is an integer, the units are assumed to be seconds.

To specify an offset from the beginning of a set of PCP archives (in archive mode) simply specify the offset as the *duration*. For example, the following entry retrieves the first sample of data at exactly 30 minutes from the beginning of a set of PCP archives:

```
-S 30min
```

To specify an offset from the end of a set of PCP archives, prefix the *duration* with a minus sign. In this case, the first sample time precedes the end of archived data by the given *duration*. For example, the following entry retrieves the first sample exactly one hour preceding the last sample in a set of PCP archives:

```
-S -1hour
```

To specify the calendar date and time (local time in the reporting timezone) for the first sample, use the **ctime(3)** syntax preceded by an "at" sign (@). For example, the following entry specifies the date and time to be used:

```
-S '@ Mon Mar 4 13:07:47 2017'
```

Note that this format corresponds to the output format of the **date** command for easy "cut and paste." However, be sure to enclose the string in quotes so it is preserved as a single argument for the PCP tool.

For more complete information on the date and time syntax, see the **PCPIntro(1)** man page.

-T *duration*

The terminate option may be used to request that the display stop at the time designated by *duration*. By default, the PCP tools keep sampling performance data indefinitely (in real-time mode) or until the end of a set of PCP archives (in archive mode). The **-T** option may be used to specify an earlier time to terminate sampling.

The interpretation for the *duration* argument in a **-T** option is the same as for the **-S** option, except for an unsigned time interval that is interpreted as being an offset from the start of the time window as defined by the default (now for real time, else start of archive set) or by a **-S** option. For example, these options define a time window that spans 45 minutes, after an initial offset (or delay) of 1 hour:

```
-S 1hour -T 45mins
```

-O *duration*

By default, samples are fetched from the start time (see the description of the **-S** option) to the terminate time (see the description of the **-T** option). The offset **-O** option allows the specification of a time between the start time and the terminate time where the tool should position its initial sample time. This option is useful when initial attention is focused at some point within a larger time window of interest, or when one PCP tool wishes to launch another PCP tool with a common current point of time within a shared time window.

The *duration* argument accepted by **-O** conforms to the same syntax and semantics as the *duration* argument for **-T**. For example, these options specify that the initial position should be the end of the time window:

```
-O -0
```

This is most useful with the **pmchart** command to display the tail-end of the history up to the end of the time window.

-A *alignment*

By default, performance data samples do not necessarily happen at any natural unit of measured time. The **-A** switch may be used to force the initial sample to be on the specified *alignment*. For example, these three options specify alignment on seconds, half hours, and whole hours:

```
-A 1sec
-A 30min
-A 1hour
```

The **-A** option advances the time to achieve the desired alignment as soon as possible after the start of the time window, whether this is the default window, or one specified with some combination of **-A** and **-O** command line options.

Obviously the time window may be overspecified by using multiple options from the set **-t, -s, -S, -T, -A,** and **-O**. Similarly, the time window may shrink to nothing by injudicious choice of options.

In all cases, the parsing of these options applies heuristics guided by the principal of "least surprise"; the time window is always well-defined (with the end never earlier than the start), but may shrink to nothing in the extreme.

5.3.3 Timezone Options

All utilities that report time of day use the local timezone by default. The following timezone options are available:

-z

Forces times to be reported in the timezone of the host that provided the metric values (the PCP collector host). When used in conjunction with **-a** and multiple archives, the convention is to use the timezone from the first named archive.

-Z *timezone*

Sets the TZ variable to a timezone string, as defined in **environ(7)**, for example, **-Z UTC** for universal time.

5.4 PCP Environment Variables

When you are using PCP tools and utilities and are calling PCP library functions, a standard set of defined environment variables are available in the `${PCP_DIR}/etc/pcp.conf` file. These variables are generally used to specify the location of various PCP pieces in the file system and may be loaded into shell scripts by sourcing the `${PCP_DIR}/etc/pcp.env` shell script. They may also be queried by C, C++, perl and python programs using the **pmGetConfig** library function. If a variable is already defined in the environment, the values in the **pcp.conf** file do not override those values; that is, the values in pcp.conf serve only as installation defaults. For additional information, see the **pcp.conf(5)**, **pcp.env(5)**, and **pmGetConfig(3)** man pages.

The following environment variables are recognized by PCP (these definitions are also available on the **PCPIntro(1)** man page):

PCP_COUNTER_WRAP

Many of the performance metrics exported from PCP agents expect that counters increase monotonically. Under some circumstances, one value of a metric may be smaller than the previously fetched value. This can happen when a counter of finite precision overflows, when the PCP agent has been reset or restarted, or when the PCP agent exports values from an underlying instrumentation that is subject to asynchronous discontinuity.

If set, the **PCP_COUNTER_WRAP** environment variable indicates that all such cases of a decreasing counter should be treated as a counter overflow; and hence the values are assumed to have wrapped once in the interval between consecutive samples. Counter wrapping was the default in versions before the PCP release 1.3.

PCP_STDERR

Specifies whether **pmprintf()** error messages are sent to standard error, an **pmconfirm** dialog box, or to a named file; see the **pmprintf(3)** man page. Messages go to standard error if **PCP_STDERR** is unset or set without a value. If this variable is set to **DISPLAY**, then messages go to an **pmconfirm** dialog box; see the **pmconfirm(1)** man page. Otherwise, the value of **PCP_STDERR** is assumed to be the name of an output file.

PMCD_CONNECT_TIMEOUT

When attempting to connect to a remote PMCD on a system that is booting or at the other end of a slow network link, some PMAPI routines could potentially block for a long time until the remote system responds. These routines abort and return an error if the connection has not been established after some specified interval has elapsed. The default interval is 5 seconds. This may be modified by setting this variable in the environment to a larger number of seconds for the desired time out. This is most useful in cases where the remote host is at the end of a slow network, requiring longer latencies to establish the connection correctly.

PMCD_PORT

This TCP/IP port is used by PMCD to create the socket for incoming connections and requests. The default is port number 44321, which you may override by setting this variable to a different port number. If a non-default port is in effect when PMCD is started, then every monitoring application connecting to that PMCD must also have this variable set in its environment before attempting a connection.

PMCD_LOCAL

This setting indicates that PMCD must only bind to the loopback interface for incoming connections and requests. In this mode, connections from remote hosts are not possible.

PMCD_RECONNECT_TIMEOUT

When a monitor or client application loses its connection to a PMCD, the connection may be reestablished by calling the **pmReconnectContext(3)** PMAPI function. However, attempts to reconnect are controlled by a back-off strategy to avoid flooding the network with reconnection requests. By default, the back-off delays are 5, 10, 20, 40, and 80 seconds for consecutive reconnection requests from a client (the last delay is repeated for any further attempts after the last delay in the list). Setting this environment variable to a comma-separated list of positive integers redefines the back-off delays. For example, setting the delays to **1,2** will back off for 1 second, then back off every 2 seconds thereafter.

PMCD_REQUEST_TIMEOUT

For monitor or client applications connected to PMCD, there is a possibility of the application hanging on a request for performance metrics or metadata or help text. These delays may become severe if the system running PMCD crashes or the network connection is lost or the network link is very slow. By setting this environment variable to a real number of seconds, requests to PMCD timeout after the specified number of seconds. The default behavior is to wait 10 seconds for a response from every PMCD for all applications.

PMLOGGER_PORT

This environment variable may be used to change the base TCP/IP port number used by **pmlogger** to create the socket to which **pmlc** instances try to connect. The default base port number is 4330. If used, this variable should be set in the environment before **pmlogger** is executed. If **pmlc** and **pmlogger** are on different hosts, then obviously **PMLOGGER_PORT** must be set to the same value in both places.

PMLOGGER_LOCAL

This environment variable indicates that **pmlogger** must only bind to the loopback interface for **pmlc** connections and requests. In this mode, **pmlc** connections from remote hosts are not possible. If used, this variable should be set in the environment before **pmlogger** is executed.

PMPROXY_PORT This environment variable may be used to change the base TCP/IP port number used by **pmproxy** to create the socket to which proxied clients connect, on their way to a distant **pmcd**.

PMPROXY_LOCAL

This setting indicates that **pmproxy** must only bind to the loopback interface for incoming connections and requests. In this mode, connections from remote hosts are not possible.

5.5 Running PCP Tools through a Firewall

In some production environments, the Performance Co-Pilot (PCP) monitoring hosts are on one side of a TCP/IP firewall, and the PCP collector hosts may be on the other side.

If the firewall service sits between the monitor and collector tools, the **pmproxy** service may be used to perform both packet forwarding and DNS proxying through the firewall; see the **pmproxy(1)** man page. Otherwise, it is necessary to arrange for packet forwarding to be enabled for those TCP/IP ports used by PCP, namely 44321 (or the value of the **PMCD_PORT** environment variable) for connections to PMCD.

5.5.1 The pmproxy service

The **pmproxy** service allows PCP clients running on hosts located on one side of a firewall to monitor remote hosts on the other side. The basic connection syntax is as follows, where *tool* is an arbitrary PCP application, typically a monitoring tool:

```
pmprobe -h remotehost@proxyhost
```

This extended host specification syntax is part of a larger set of available extensions to the basic host naming syntax - refer to the **PCPIntro(1)** man page for further details.

5.6 Transient Problems with Performance Metric Values

Sometimes the values for a performance metric as reported by a PCP tool appear to be incorrect. This is typically caused by transient conditions such as metric wraparound or time skew, described below. These conditions result from design decisions that are biased in favor of lightweight protocols and minimal resource demands for PCP components.

In all cases, these events are expected to occur infrequently, and should not persist beyond a few samples.

5.6.1 Performance Metric Wraparound

Performance metrics are usually expressed as numbers with finite precision. For metrics that are cumulative counters of events or resource consumption, the value of the metric may occasionally overflow the specified range and wraparound to zero.

Because the value of these counter metrics is computed from the rate of change with respect to the previous sample, this may result in a transient condition where the rate of change is an unknown value. If the **PCP_COUNTER_WRAP** environment variable is set, this condition is treated as an overflow, and speculative rate calculations are made. In either case, the correct rate calculation for the metric returns with the next sample.

5.6.2 Time Dilation and Time Skew

If a PMDA is tardy in returning results, or the PCP monitoring tool is connected to PMCD via a slow or congested network, an error might be introduced in rate calculations due to a difference between the time the metric was sampled and the time PMCD sends the result to the monitoring tool.

In practice, these errors are usually so small as to be insignificant, and the errors are self-correcting (not cumulative) over consecutive samples.

A related problem may occur when the system time is not synchronized between multiple hosts, and the time stamps for the results returned from PMCD reflect the skew in the system times. In this case, it is recommended that NTP

(network time protocol) be used to keep the system clocks on the collector systems synchronized; for information on NTP refer to the **ntpd(1)** man page.

CHAPTER

SIX

MONITORING SYSTEM PERFORMANCE

Contents

- *Monitoring System Performance*
  - *The pmstat Command*
  - *The pmrep Command*
  - *The pmval Command*
  - *The pminfo Command*
  - *The pmstore Command*

This chapter describes the performance monitoring tools available in Performance Co-Pilot (PCP). This product provides a group of commands and tools for measuring system performance. Each tool is described completely by its own man page. The man pages are accessible through the **man** command. For example, the man page for the tool **pmrep** is viewed by entering the following command:

```
man pmrep
```

The following major sections are covered in this chapter:

Section 4.1, "*The pmstat Command*", discusses **pmstat**, a utility that provides a periodic one-line summary of system performance.

Section 4.2, "*The pmrep Command*", discusses **pmrep**, a utility that shows the current values for named performance metrics.

Section 4.3, "*The pmval Command*", describes **pmval**, a utility that displays performance metrics in a textual format.

Section 4.4, "*The pminfo Command*", describes **pminfo**, a utility that displays information about performance metrics.

Section 4.5, "*The pmstore Command*", describes the use of the **pmstore** utility to arbitrarily set or reset selected performance metric values.

The following sections describe the various graphical and text-based PCP tools used to monitor local or remote system performance.

6.1 The pmstat Command

The **pmstat** command provides a periodic, one-line summary of system performance. This command is intended to monitor system performance at the highest level, after which other tools may be used for examining subsystems to observe potential performance problems in greater detail. After entering the **pmstat** command, you see output similar to the following, with successive lines appearing periodically:

```
pmstat
@ Thu Aug 15 09:25:56 2017
 loadavg                     memory      swap        io     system         cpu
   1 min   swpd   free   buff  cache   pi   po   bi   bo   in   cs  us  sy  id
    1.29 833960  5614m 144744 265824    0    0    0 1664  13K  23K   6   7  81
    1.51 833956  5607m 144744 265712    0    0    0 1664  13K  24K   5   7  83
    1.55 833956  5595m 145196 271908    0    0  14K 1056  13K  24K   7   7  74
```

An additional line of output is added every five seconds. The **-t** *interval* option may be used to vary the update interval (i.e. the sampling interval).

The output from **pmstat** is directed to standard output, and the columns in the report are interpreted as follows:

loadavg

The 1-minute load average (runnable processes).

memory

The swpd column indicates average swap space used during the interval (all columns reported in Kbytes unless otherwise indicated). The **free** column indicates average free memory during the interval. The **buff** column indicates average buffer memory in use during the interval. The **cache** column indicates average cached memory in use during the interval.

swap

Reports the average number of pages that are paged-in (**pi**) and paged-out (**po**) per second during the interval. It is normal for the paged-in values to be non-zero, but the system is suffering memory stress if the paged-out values are non-zero over an extended period.

io

The **bi** and **bo** columns indicate the average rate per second of block input and block output operations respectfully, during the interval. These rates are independent of the I/O block size. If the values become large, they are reported as thousands of operations per second (K suffix) or millions of operations per second (M suffix).

system

Context switch rate (**cs**) and interrupt rate (**in**). Rates are expressed as average operations per second during the interval. Note that the interrupt rate is normally at least HZ (the clock interrupt rate, and **kernel.all.hz** metric) interrupts per second.

cpu

Percentage of CPU time spent executing user code (**us**), system and interrupt code (**sy**), idle loop (**id**).

As with most PCP utilities, real-time metric, and archive logs are interchangeable.

For example, the following command uses a local system PCP archive log *20170731* and the timezone of the host (**smash**) from which performance metrics in the archive were collected:

```
pmstat -a ${PCP_LOG_DIR}/pmlogger/smash/20170731 -t 2hour -A 1hour -z
Note: timezone set to local timezone of host "smash"
@ Wed Jul 31 10:00:00 2017
```

(continues on next page)

(continued from previous page)

```
 loadavg                     memory      swap        io     system         cpu
   1 min   swpd   free   buff  cache   pi   po   bi   bo   in   cs  us  sy  id
    3.90  24648  6234m 239176  2913m    ?    ?    ?    ?    ?    ?   ?   ?   ?
    1.72  24648  5273m 239320  2921m    0    0    4   86  11K  19K   5   5  84
    3.12  24648  5194m 241428  2969m    0    0    0   84  10K  19K   5   5  85
    1.97  24644  4945m 244004  3146m    0    0    0   84  10K  19K   5   5  84
    3.82  24640  4908m 244116  3147m    0    0    0   83  10K  18K   5   5  85
    3.38  24620  4860m 244116  3148m    0    0    0   83  10K  18K   5   4  85
    2.89  24600  4804m 244120  3149m    0    0    0   83  10K  18K   5   4  85
pmFetch: End of PCP archive log
```

For complete information on **pmstat** usage and command line options, see the **pmstat(1)** man page.

6.2 The pmrep Command

The **pmrep** command displays performance metrics in ASCII tables, suitable for export into databases or report generators. It is a flexible command. For example, the following command provides continuous memory statistics on a host named **surf**:

```
pmrep -p -h surf kernel.all.load kernel.all.pswitch
          k.a.load  k.a.load  k.a.load  k.a.pswitch
          1 minute  5 minute  15 minut
                                          count/s
10:41:37     0.160     0.170     0.180          N/A
10:41:38     0.160     0.170     0.180     1427.016
10:41:39     0.160     0.170     0.180     2129.040
10:41:40     0.160     0.170     0.180     5335.163
10:41:41     0.160     0.170     0.180      723.125
10:41:42     0.140     0.160     0.180      591.859
```

See the **pmrep(1)** man page for more information.

6.3 The pmval Command

The **pmval** command dumps the current values for the named performance metrics. For example, the following command reports the value of performance metric **proc.nprocs** once per second (by default), and produces output similar to this:

```
pmval proc.nprocs
metric:    proc.nprocs
host:      localhost
semantics: instantaneous value
units:     none
samples:   all
interval:  1.00 sec
         81
         81
         82
         81
```

In this example, the number of running processes was reported once per second.

Where the semantics of the underlying performance metrics indicate that it would be sensible, **pmval** reports the rate of change or resource utilization.

For example, the following command reports idle processor utilization for each of four CPUs on the remote host **dove**, each five seconds apart, producing output of this form:

```
pmval -h dove -t 5sec -s 4 kernel.percpu.cpu.idle
metric:    kernel.percpu.cpu.idle
host:      dove
semantics: cumulative counter (converting to rate)
units:     millisec (converting to time utilization)
samples:   4
interval:  5.00 sec

cpu:1.1.0.a cpu:1.1.0.c cpu:1.1.1.a cpu:1.1.1.c
     1.000       0.9998      0.9998      1.000
     1.000       0.9998      0.9998      1.000
     0.8989      0.9987      0.9997      0.9995
     0.9568      0.9998      0.9996      1.000
```

Similarly, the following command reports disk I/O read rate every minute for just the disk **/dev/disk1**, and produces output similar to the following:

```
pmval -t 1min -i disk1 disk.dev.read
metric:    disk.dev.read
host:      localhost
semantics: cumulative counter (converting to rate)
units:     count (converting to count / sec)
samples:   indefinite
interval:  60.00 sec
        disk1
         33.67
         48.71
         52.33
         11.33
         2.333
```

The **-r** flag may be used to suppress the rate calculation (for metrics with counter semantics) and display the raw values of the metrics.

In the example below, manipulation of the time within the archive is achieved by the exchange of time control messages between **pmval** and **pmtime**.

```
pmval -g -a ${PCP_LOG_DIR}/pmlogger/myserver/20170731 kernel.all.load
```

The **pmval** command is documented by the **pmval(1)** man page, and annotated examples of the use of **pmval** can be found in the *PCP Tutorials and Case Studies* companion document.

6.4 The pminfo Command

The **pminfo** command displays various types of information about performance metrics available through the Performance Co-Pilot (PCP) facilities.

The **-T** option is extremely useful; it provides help text about performance metrics:

```
pminfo -T mem.util.cached
mem.util.cached
Help:
Memory used by the page cache, including buffered file data.
This is in-memory cache for files read from the disk (the pagecache)
but doesn't include SwapCached.
```

The **-t** option displays the one-line help text associated with the selected metrics. The **-T** option prints more verbose help text.

Without any options, **pminfo** verifies that the specified metrics exist in the namespace, and echoes those names. Metrics may be specified as arguments to **pminfo** using their full metric names. For example, this command returns the following response:

```
pminfo hinv.ncpu network.interface.total.bytes
hinv.ncpu
network.interface.total.bytes
```

A group of related metrics in the namespace may also be specified. For example, to list all of the **hinv** metrics you would use this command:

```
pminfo hinv
hinv.physmem
hinv.pagesize
hinv.ncpu
hinv.ndisk
hinv.nfilesys
hinv.ninterface
hinv.nnode
hinv.machine
hinv.map.scsi
hinv.map.cpu_num
hinv.map.cpu_node
hinv.map.lvname
hinv.cpu.clock
hinv.cpu.vendor
hinv.cpu.model
hinv.cpu.stepping
hinv.cpu.cache
hinv.cpu.bogomips
```

If no metrics are specified, **pminfo** displays the entire collection of metrics. This can be useful for searching for metrics, when only part of the full name is known. For example, this command returns the following response:

```
pminfo | grep nfs
nfs.client.calls
nfs.client.reqs
nfs.server.calls
nfs.server.reqs
nfs3.client.calls
```

(continues on next page)

(continued from previous page)

```
nfs3.client.reqs
nfs3.server.calls
nfs3.server.reqs
nfs4.client.calls
nfs4.client.reqs
nfs4.server.calls
nfs4.server.reqs
```

The **-d** option causes **pminfo** to display descriptive information about metrics (refer to the **pmLookupDesc(3)** man page for an explanation of this metadata information). The following command and response show use of the **-d** option:

```
pminfo -d proc.nprocs disk.dev.read filesys.free
proc.nprocs
    Data Type: 32-bit unsigned int  InDom: PM_INDOM_NULL 0xffffffff
    Semantics: instant  Units: none

disk.dev.read
    Data Type: 32-bit unsigned int  InDom: 60.1 0xf000001
    Semantics: counter  Units: count

filesys.free
    Data Type: 64-bit unsigned int  InDom: 60.5 0xf000005
    Semantics: instant  Units: Kbyte
```

The **-l** option causes **pminfo** to display labels about metrics (refer to the **pmLookupLabels(3)** man page for an explanation of this metadata information). If the metric has an instance domain, the labels associated with each instance of the metric is printed. The following command and response show use of the **-l** option:

```
pminfo -l -h shard kernel.pernode.cpu.user
kernel.percpu.cpu.sys
    inst [0 or "cpu0"] labels
{"agent":"linux","cpu":0,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [1 or "cpu1"] labels
{"agent":"linux","cpu":1,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [2 or "cpu2"] labels
{"agent":"linux","cpu":2,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [3 or "cpu3"] labels
{"agent":"linux","cpu":3,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [4 or "cpu4"] labels
{"agent":"linux","cpu":4,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [5 or "cpu5"] labels
{"agent":"linux","cpu":5,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [6 or "cpu6"] labels
{"agent":"linux","cpu":6,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
    inst [7 or "cpu7"] labels
{"agent":"linux","cpu":7,"device_type":"cpu","domainname":"acme.com","groupid":1000,
↪"hostname":"shard","indom_name":"per cpu","userid":1000}
```

The **-f** option to **pminfo** forces the current value of each named metric to be fetched and printed. In the example below,

all metrics in the group **hinv** are selected:

```
pminfo -f hinv
hinv.physmem
    value 15701

hinv.pagesize
    value 16384

hinv.ncpu
    value 4

hinv.ndisk
    value 6

hinv.nfilesys
    value 2

hinv.ninterface
    value 8

hinv.nnode
    value 2

hinv.machine
    value "IP35"

hinv.map.cpu_num
    inst [0 or "cpu:1.1.0.a"] value 0
    inst [1 or "cpu:1.1.0.c"] value 1
    inst [2 or "cpu:1.1.1.a"] value 2
    inst [3 or "cpu:1.1.1.c"] value 3

hinv.map.cpu_node
    inst [0 or "node:1.1.0"] value "/dev/hw/module/001c01/slab/0/node"
    inst [1 or "node:1.1.1"] value "/dev/hw/module/001c01/slab/1/node"

hinv.cpu.clock
    inst [0 or "cpu:1.1.0.a"] value 800
    inst [1 or "cpu:1.1.0.c"] value 800
    inst [2 or "cpu:1.1.1.a"] value 800
    inst [3 or "cpu:1.1.1.c"] value 800

hinv.cpu.vendor
    inst [0 or "cpu:1.1.0.a"] value "GenuineIntel"
    inst [1 or "cpu:1.1.0.c"] value "GenuineIntel"
    inst [2 or "cpu:1.1.1.a"] value "GenuineIntel"
    inst [3 or "cpu:1.1.1.c"] value "GenuineIntel"

hinv.cpu.model
    inst [0 or "cpu:1.1.0.a"] value "0"
    inst [1 or "cpu:1.1.0.c"] value "0"
    inst [2 or "cpu:1.1.1.a"] value "0"
    inst [3 or "cpu:1.1.1.c"] value "0"

hinv.cpu.stepping
    inst [0 or "cpu:1.1.0.a"] value "6"
    inst [1 or "cpu:1.1.0.c"] value "6"
```

(continues on next page)

(continued from previous page)

```
    inst [2 or "cpu:1.1.1.a"] value "6"
    inst [3 or "cpu:1.1.1.c"] value "6"

hinv.cpu.cache
    inst [0 or "cpu:1.1.0.a"] value 0
    inst [1 or "cpu:1.1.0.c"] value 0
    inst [2 or "cpu:1.1.1.a"] value 0
    inst [3 or "cpu:1.1.1.c"] value 0

hinv.cpu.bogomips
    inst [0 or "cpu:1.1.0.a"] value 1195.37
    inst [1 or "cpu:1.1.0.c"] value 1195.37
    inst [2 or "cpu:1.1.1.a"] value 1195.37
    inst [3 or "cpu:1.1.1.c"] value 1195.37
```

The **-h** option directs **pminfo** to retrieve information from the specified host. If the metric has an instance domain, the value associated with each instance of the metric is printed:

```
pminfo -h dove -f filesys.mountdir
filesys.mountdir
    inst [0 or "/dev/xscsi/pci00.01.0/target81/lun0/part3"] value "/"
    inst [1 or "/dev/xscsi/pci00.01.0/target81/lun0/part1"] value "/boot/efi"
```

The **-m** option prints the Performance Metric Identifiers (PMIDs) of the selected metrics. This is useful for finding out which PMDA supplies the metric. For example, the output below identifies the PMDA supporting domain 4 (the leftmost part of the PMID) as the one supplying information for the metric **environ.extrema.mintemp**:

```
pminfo -m environ.extrema.mintemp
environ.extrema.mintemp PMID: 4.0.3
```

The **-v** option verifies that metric definitions in the PMNS correspond with supported metrics, and checks that a value is available for the metric. Descriptions and values are fetched, but not printed. Only errors are reported.

Complete information on the **pminfo** command is found in the **pminfo(1)** man page. There are further examples of the use of **pminfo** in the *PCP Tutorials and Case Studies*.

6.5 The pmstore Command

From time to time you may wish to change the value of a particular metric. Some metrics are counters that may need to be reset, and some are simply control variables for agents that collect performance metrics. When you need to change the value of a metric for any reason, the command to use is **pmstore**.

Note: For obvious reasons, the ability to arbitrarily change the value of a performance metric is not supported. Rather, PCP collectors selectively allow some metrics to be modified in a very controlled fashion.

The basic syntax of the command is as follows:

```
pmstore metricname value
```

There are also command line flags to further specify the action. For example, the **-i** option restricts the change to one or more instances of the performance metric.

The *value* may be in one of several forms, according to the following rules:

1. If the metric has an integer type, then value should consist of an optional leading hyphen, followed either by decimal digits or "0x" and some hexadecimal digits; "0X" is also acceptable instead of "0x."
2. If the metric has a floating point type, then value should be in the form of an integer (described above), a fixed point number, or a number in scientific notation.
3. If the metric has a string type, then value is interpreted as a literal string of ASCII characters.
4. If the metric has an aggregate type, then an attempt is made to interpret value as an integer, a floating point number, or a string. In the first two cases, the minimal word length encoding is used; for example, "123" would be interpreted as a four-byte aggregate, and "0x100000000" would be interpreted as an eight-byte aggregate.

The following example illustrates the use of **pmstore** to enable performance metrics collection in the **txmon** PMDA (see `${PCP_PMDAS_DIR}/txmon` for the source code of the txmon PMDA). When the metric **txmon.control.level** has the value 0, no performance metrics are collected. Values greater than 0 enable progressively more verbose instrumentation.

```
pminfo -f txmon.count
txmon.count
No value(s) available!
pmstore txmon.control.level 1
txmon.control.level old value=0 new value=1
pminfo -f txmon.count
txmon.count
       inst [0 or "ord-entry"] value 23
       inst [1 or "ord-enq"] value 11
       inst [2 or "ord-ship"] value 10
       inst [3 or "part-recv"] value 3
       inst [4 or "part-enq"] value 2
       inst [5 or "part-used"] value 1
       inst [6 or "b-o-m"] value 0
```

For complete information on **pmstore** usage and syntax, see the **pmstore(1)** man page.

CHAPTER

SEVEN

PERFORMANCE METRICS INFERENCE ENGINE

Contents

- *Performance Metrics Inference Engine*
  - *Introduction to pmie*
  - *Basic pmie Usage*
    - *pmie use of PCP services*
    - *Simple pmie Usage*
    - *Complex pmie Examples*
  - *Specification Language for pmie*
    - *Basic pmie Syntax*
      - *Lexical Elements*
      - *Comments*
      - *Macros*
      - *Units*
    - *Setting Evaluation Frequency*
    - *pmie Metric Expressions*
    - *pmie Rate Conversion*
    - *pmie Arithmetic Expressions*
    - *pmie Logical Expressions*
      - *Logical Constants*
      - *Relational Expressions*
      - *Boolean Expressions*
      - *Quantification Operators*
    - *pmie Rule Expressions*
    - *pmie Intrinsic Operators*
      - *Arithmetic Aggregation*
      - *The rate Operator*

- Transitional Operators
- pmie Examples
- Developing and Debugging pmie Rules
- Caveats and Notes on pmie
  - Performance Metrics Wraparound
  - pmie Sample Intervals
  - pmie Instance Names
  - pmie Error Detection
- Creating pmie Rules with pmieconf
- Management of pmie Processes
  - Add a pmie crontab Entry
  - Global Files and Directories
  - pmie Instances and Their Progress

The Performance Metrics Inference Engine (**pmie**) is a tool that provides automated monitoring of, and reasoning about, system performance within the Performance Co-Pilot (PCP) framework.

The major sections in this chapter are as follows:

Section 5.1, "*Introduction to pmie*", provides an introduction to the concepts and design of **pmie**.

Section 5.2, "*Basic pmie Usage*", describes the basic syntax and usage of **pmie**.

Section 5.3, "*Specification Language for pmie*", discusses the complete **pmie** rule specification language.

Section 5.4, "*pmie Examples*", provides an example, covering several common performance scenarios.

Section 5.5, "*Developing and Debugging pmie Rules*", presents some tips and techniques for **pmie** rule development.

Section 5.6, "*Caveats and Notes on pmie*", presents some important information on using **pmie**.

Section 5.7, "*Creating pmie Rules with pmieconf*", describes how to use the **pmieconf** command to generate **pmie** rules.

Section 5.8, "*Management of pmie Processes*", provides support for running **pmie** as a daemon.

7.1 Introduction to pmie

Automated reasoning within Performance Co-Pilot (PCP) is provided by the Performance Metrics Inference Engine, (**pmie**), which is an applied artificial intelligence application.

The **pmie** tool accepts expressions describing adverse performance scenarios, and periodically evaluates these against streams of performance metric values from one or more sources. When an expression is found to be true, **pmie** is able to execute arbitrary actions to alert or notify the system administrator of the occurrence of an adverse performance scenario. These facilities are very general, and are designed to accommodate the automated execution of a mixture of generic and site-specific performance monitoring and control functions.

The stream of performance metrics to be evaluated may be from one or more hosts, or from one or more PCP archive logs. In the latter case, **pmie** may be used to retrospectively identify adverse performance conditions.

Using **pmie**, you can filter, interpret, and reason about the large volume of performance data made available from PCP collector systems or PCP archives.

Typical **pmie** uses include the following:

- Automated real-time monitoring of a host, a set of hosts, or client-server pairs of hosts to raise operational alarms when poor performance is detected in a production environment
- Nightly processing of archive logs to detect and report performance regressions, or quantify quality of service for service level agreements or management reports, or produce advance warning of pending performance problems
- Strategic performance management, for example, detection of slightly abnormal to chronic system behavior, trend analysis, and capacity planning

The **pmie** expressions are described in a language with expressive power and operational flexibility. It includes the following operators and functions:

- Generalized predicate-action pairs, where a predicate is a logical expression over the available performance metrics, and the action is arbitrary. Predefined actions include the following:
  - Launch a visible alarm with **pmconfirm**; see the **pmconfirm(1)** man page.
  - Post an entry to the system log file; see the **syslog(3)** man page.
  - Post an entry to the PCP noticeboard file `${PCP_LOG_DIR}/NOTICES`; see the **pmpost(1)** man page.
  - Execute a shell command or script, for example, to send e-mail, initiate a pager call, warn the help desk, and so on.
  - Echo a message on standard output; useful for scripts that generate reports from retrospective processing of PCP archive logs.
- Arithmetic and logical expressions in a C-like syntax.
- Expression groups may have an independent evaluation frequency, to support both short-term and long-term monitoring.
- Canonical scale and rate conversion of performance metric values to provide sensible expression evaluation.
- Aggregation functions of **sum, avg, min**, and **max**, that may be applied to collections of performance metrics values clustered over multiple hosts, or multiple instances, or multiple consecutive samples in time.
- Universal and existential quantification, to handle expressions of the form "for every. . . ." and "at least one. . . ".
- Percentile aggregation to handle statistical outliers, such as "for at least 80% of the last 20 samples, . . . ".
- Macro processing to expedite repeated use of common subexpressions or specification components.
- Transparent operation against either live-feeds of performance metric values from PMCD on one or more hosts, or against PCP archive logs of previously accumulated performance metric values.

The power of **pmie** may be harnessed to automate the most common of the deterministic system management functions that are responses to changes in system performance. For example, disable a batch stream if the DBMS transaction commit response time at the ninetieth percentile goes over two seconds, or stop accepting uploads and send e-mail to the *sysadmin* alias if free space in a storage system falls below five percent.

Moreover, the power of **pmie** can be directed towards the exceptional and sporadic performance problems. For example, if a network packet storm is expected, enable IP header tracing for ten seconds, and send e-mail to advise that data has been collected and is awaiting analysis. Or, if production batch throughput falls below 50 jobs per minute, activate a pager to the systems administrator on duty.

Obviously, **pmie** customization is required to produce meaningful filtering and actions in each production environment. The **pmieconf** tool provides a convenient customization method, allowing the user to generate parameterized **pmie** rules for some of the more common performance scenarios.

7.2 Basic pmie Usage

This section presents and explains some basic examples of **pmie** usage. The **pmie** tool accepts the common PCP command line arguments, as described in Chapter 3, *Common Conventions and Arguments*. In addition, **pmie** accepts the following command line arguments:

-d	Enables interactive debug mode.
-v	Verbose mode: expression values are displayed.
-V	Verbose mode: annotated expression values are displayed.
-W	When-verbose mode: when a condition is true, the satisfying expression bindings are displayed.

One of the most basic invocations of this tool is this form:

```
pmie filename
```

In this form, the expressions to be evaluated are read from *filename*. In the absence of a given *filename*, expressions are read from standard input, which may be your system keyboard.

7.2.1 pmie use of PCP services

Before you use **pmie**, it is strongly recommended that you familiarize yourself with the concepts from the Section 1.2, "*Conceptual Foundations*". The discussion in this section serves as a very brief review of these concepts.

PCP makes available thousands of performance metrics that you can use when formulating expressions for **pmie** to evaluate. If you want to find out which metrics are currently available on your system, use this command:

```
pminfo
```

Use the **pmie** command line arguments to find out more about a particular metric. In *Example 5.1. pmie with the -f Option*, to fetch new metric values from host **dove**, you use the **-f** flag:

Example 5.1. pmie with the -f Option

```
pminfo -f -h dove disk.dev.total
```

This produces the following response:

```
disk.dev.total
    inst [0 or "xscsi/pci00.01.0/target81/lun0/disc"] value 131233
    inst [4 or "xscsi/pci00.01.0/target82/lun0/disc"] value 4
    inst [8 or "xscsi/pci00.01.0/target83/lun0/disc"] value 4
    inst [12 or "xscsi/pci00.01.0/target84/lun0/disc"] value 4
    inst [16 or "xscsi/pci00.01.0/target85/lun0/disc"] value 4
    inst [18 or "xscsi/pci00.01.0/target86/lun0/disc"] value 4
```

This reveals that on the host **dove**, the metric **disk.dev.total** has six instances, one for each disk on the system.

Use the following command to request help text (specified with the **-T** flag) to provide more information about performance metrics:

```
pminfo -T network.interface.in.packets
```

The metadata associated with a performance metric is used by **pmie** to determine how the value should be interpreted. You can examine the descriptor that encodes the metadata by using the **-d** flag for **pminfo**, as shown in *Example 5.2. pmie with the -d and -h Options* :

Example 5.2. pmie with the -d and -h Options

```
pminfo -d -h somehost mem.util.cached kernel.percpu.cpu.user
```

In response, you see output similar to this:

```
mem.util.cached
    Data Type: 64-bit unsigned int  InDom: PM_INDOM_NULL 0xffffffff
    Semantics: instant  Units: Kbyte

kernel.percpu.cpu.user
    Data Type: 64-bit unsigned int  InDom: 60.0 0xf000000
    Semantics: counter  Units: millisec
```

Note: A cumulative counter such as **kernel.percpu.cpu.user** is automatically converted by **pmie** into a rate (measured in events per second, or count/second), while instantaneous values such as **mem.util.cached** are not subjected to rate conversion. Metrics with an instance domain (**InDom** in the **pminfo** output) of **PM_INDOM_NULL** are singular and always produce one value per source. However, a metric like **kernel.percpu.cpu.user** has an instance domain, and may produce multiple values per source (in this case, it is one value for each configured CPU).

7.2.2 Simple pmie Usage

Example 5.3. pmie with the -v Option directs the inference engine to evaluate and print values (specified with the **-v** flag) for a single performance metric (the simplest possible expression), in this case **disk.dev.total**, collected from the local PMCD:

Example 5.3. pmie with the -v Option

```
pmie -v
iops = disk.dev.total;
Ctrl+D
iops:      ?      ?
iops:   14.4      0
iops:   25.9  0.112
iops:   12.2      0
iops:   12.3   64.1
iops:  8.594  52.17
iops:  2.001  71.64
```

On this system, there are two disk spindles, hence two values of the expression **iops** per sample. Notice that the values for the first sample are unknown (represented by the question marks [?] in the first line of output), because rates can be computed only when at least two samples are available. The subsequent samples are produced every ten seconds by default. The second sample reports that during the preceding ten seconds there was an average of 14.4 transfers per second on one disk and no transfers on the other disk.

Rates are computed using time stamps delivered by PMCD. Due to unavoidable inaccuracy in the actual sampling time (the sample interval is not exactly 10 seconds), you may see more decimal places in values than you expect. Notice, however, that these errors do not accumulate but cancel each other out over subsequent samples.

In *Example 5.3. pmie with the -v Option*, the expression to be evaluated was entered using the keyboard, followed by the end-of-file character [**Ctrl+D**]. Usually, it is more convenient to enter expressions into a file (for example, **myrules**) and ask **pmie** to read the file. Use this command syntax:

```
pmie -v myrules
```

Please refer to the **pmie(1)** man page for a complete description of **pmie** command line options.

7.2.3 Complex pmie Examples

This section illustrates more complex **pmie** expressions of the specification language. Section 5.3, "*Specification Language for pmie*", provides a complete description of the **pmie** specification language.

The following arithmetic expression computes the percentage of write operations over the total number of disk transfers.

```
(disk.all.write / disk.all.total) * 100;
```

The **disk.all** metrics are singular, so this expression produces exactly one value per sample, independent of the number of disk devices.

Note: If there is no disk activity, **disk.all.total** will be zero and **pmie** evaluates this expression to be not a number. When **-v** is used, any such values are displayed as question marks.

The following logical expression has the value **true** or **false** for each disk:

```
disk.dev.total > 10 &&
disk.dev.write > disk.dev.read;
```

The value is true if the number of writes exceeds the number of reads, and if there is significant disk activity (more than 10 transfers per second). *Example 5.4. Printed pmie Output* demonstrates a simple action:

Example 5.4. Printed pmie Output

```
some_inst disk.dev.total > 60
          -> print "[%i] high disk i/o";
```

This prints a message to the standard output whenever the total number of transfers for some disk (**some_inst**) exceeds 60 transfers per second. The **%i** (instance) in the message is replaced with the name(s) of the disk(s) that caused the logical expression to be **true**.

Using **pmie** to evaluate the above expressions every 3 seconds, you see output similar to *Example 5.5. Labelled pmie Output*. Notice the introduction of labels for each **pmie** expression.

Example 5.5. Labelled pmie Output

```
pmie -v -t 3sec
pct_wrt = (disk.all.write / disk.all.total) * 100;
busy_wrt = disk.dev.total > 10 &&
           disk.dev.write > disk.dev.read;
busy = some_inst disk.dev.total > 60
           -> print "[%i] high disk i/o ";
Ctrl+D
pct_wrt:       ?
busy_wrt:      ?      ?
busy:          ?

pct_wrt:   18.43
busy_wrt:  false  false
busy:      false

Mon Aug  5 14:56:08 2012: [disk2] high disk i/o
```

(continues on next page)

(continued from previous page)

```
pct_wrt:   10.83
busy_wrt:  false  false
busy:      true

pct_wrt:   19.85
busy_wrt:   true  false
busy:      false

pct_wrt:       ?
busy_wrt:  false  false
busy:      false

Mon Aug  5 14:56:17 2012: [disk1] high disk i/o [disk2] high disk i/o
pct_wrt:   14.8
busy_wrt:  false  false
busy:   true
```

The first sample contains unknowns, since all expressions depend on computing rates. Also notice that the expression **pct_wrt** may have an undefined value whenever all disks are idle, as the denominator of the expression is zero. If one or more disks is busy, the expression **busy** is true, and the message from the **print** in the action part of the rule appears (before the **-v** values).

7.3 Specification Language for pmie

This section describes the complete syntax of the **pmie** specification language, as well as macro facilities and the issue of sampling and evaluation frequency. The reader with a preference for learning by example may choose to skip this section and go straight to the examples in Section 5.4, "*pmie Examples*".

Complex expressions are built up recursively from simple elements:

1. Performance metric values are obtained from PMCD for real-time sources, otherwise from PCP archive logs.
2. Metrics values may be combined using arithmetic operators to produce arithmetic expressions.
3. Arithmetic expressions may be compared using relational operators to produce logical expressions.
4. Logical expressions may be combined using Boolean operators, including powerful quantifiers.
5. Aggregation operators may be used to compute summary expressions, for either arithmetic or logical operands.
6. The final logical expression may be used to initiate a sequence of actions.

7.3.1 Basic pmie Syntax

The **pmie** rule specification language supports a number of basic syntactic elements.

Lexical Elements

All **pmie** expressions are composed of the following lexical elements:

Identifier

Begins with an alphabetic character (either upper or lowercase), followed by zero or more letters, the numeric digits, and the special characters period (.) and underscore (_), as shown in the following example:

```
x, disk.dev.total and my_stuff
```

As a special case, an arbitrary sequence of letters enclosed by apostrophes (‘) is also interpreted as an *identifier*; for example:

```
'vms$slow_response'
```

Keyword

The aggregate operators, units, and predefined actions are represented by keywords; for example, **some_inst**, **print**, and **hour**.

Numeric constant

Any likely representation of a decimal integer or floating point number; for example, 124, 0.05, and -45.67

String constant

An arbitrary sequence of characters, enclosed by double quotation marks (**“x”**).

Within quotes of any sort, the backslash () may be used as an escape character as shown in the following example:

```
"A \"gentle\" reminder"
```

Comments

Comments may be embedded anywhere in the source, in either of these forms:

/* text */	Comment, optionally spanning multiple lines, with no nesting of comments.
// text	Comment from here to the end of the line.

Macros

When they are fully specified, expressions in **pmie** tend to be verbose and repetitive. The use of macros can reduce repetition and improve readability and modularity. Any statement of the following form associates the macro name **identifier** with the given string constant.

```
identifier = "string";
```

Any subsequent occurrence of the macro name **identifier** is replaced by the string most recently associated with a macro definition for **identifier**.

```
$identifier
```

For example, start with the following macro definition:

```
disk = "disk.all";
```

You can then use the following syntax:

```
pct_wrt = ($disk.write / $disk.total) * 100;
```

Note: Macro expansion is performed before syntactic parsing; so macros may only be assigned constant string values.

Units

The inference engine converts all numeric values to canonical units (seconds for time, bytes for space, and events for count). To avoid surprises, you are encouraged to specify the units for numeric constants. If units are specified, they are checked for dimension compatibility against the metadata for the associated performance metrics.

The syntax for a **units** specification is a sequence of one or more of the following keywords separated by either a space or a slash (/), to denote per: **byte, KByte, MByte, GByte, TByte, nsec, nanosecond, usec, microsecond, msec, millisecond, sec, second, min, minute, hour, count, Kcount, Mcount, Gcount,** or **Tcount**. Plural forms are also accepted.

The following are examples of units usage:

```
disk.dev.blktotal > 1 Mbyte / second;
mem.util.cached < 500 Kbyte;
```

Note: If you do not specify the units for numeric constants, it is assumed that the constant is in the canonical units of seconds for time, bytes for space, and events for count, and the dimensionality of the constant is assumed to be correct. Thus, in the following expression, the **500** is interpreted as 500 bytes.

```
mem.util.cached < 500
```

7.3.2 Setting Evaluation Frequency

The identifier name **delta** is reserved to denote the interval of time between consecutive evaluations of one or more expressions. Set **delta** as follows:

```
delta = number [units];
```

If present, **units** must be one of the time units described in the preceding section. If absent, **units** are assumed to be **seconds**. For example, the following expression has the effect that any subsequent expressions (up to the next expression that assigns a value to **delta**) are scheduled for evaluation at a fixed frequency, once every five minutes.

```
delta = 5 min;
```

The default value for **delta** may be specified using the **-t** command line option; otherwise **delta** is initially set to be 10 seconds.

7.3.3 pmie Metric Expressions

The performance metrics namespace (PMNS) provides a means of naming performance metrics, for example, **disk.dev.read**. PCP allows an application to retrieve one or more values for a performance metric from a designated source (a collector host running PMCD, or a set of PCP archive logs). To specify a single value for some performance metric requires the metric name to be associated with all three of the following:

1. A particular host (or source of metrics values)
2. A particular instance (for metrics with multiple values)
3. A sample time

The permissible values for hosts are the range of valid hostnames as provided by Internet naming conventions.

The names for instances are provided by the Performance Metrics Domain Agents (PMDA) for the instance domain associated with the chosen performance metric.

The sample time specification is defined as the set of natural numbers 0, 1, 2, and so on. A number refers to one of a sequence of sampling events, from the current sample 0 to its predecessor 1, whose predecessor was 2, and so on. This scheme is illustrated by the time line shown in *Figure 5.1. Sampling Time Line*.

Fig. 1: Figure 5.1. Sampling Time Line

Each sample point is assumed to be separated from its predecessor by a constant amount of real time, the **delta**. The most recent sample point is always zero. The value of **delta** may vary from one expression to the next, but is fixed for each expression; for more information on the sampling interval, see Section 5.3.2, "*Setting Evaluation Frequency*".

For **pmie**, a metrics expression is the name of a metric, optionally qualified by a host, instance and sample time specification. Special characters introduce the qualifiers: colon (**:**) for hosts, hash or pound sign (**#**) for instances, and at (**@**) for sample times. The following expression refers to the previous value (**@1**) of the counter for the disk read operations associated with the disk instance **#disk1** on the host **moomba**.

```
disk.dev.read :moomba #disk1 @1
```

In fact, this expression defines a point in the three-dimensional (3D) parameter space of {**host**} x {**instance**} x {**sample time**} as shown in *Figure 5.2. Three-Dimensional Parameter Space*.

Fig. 2: Figure 5.2. Three-Dimensional Parameter Space

A metric expression may also identify sets of values corresponding to one-, two-, or three-dimensional slices of this space, according to the following rules:

1. A metric expression consists of a PCP metric name, followed by optional host specifications, followed by optional instance specifications, and finally, optional sample time specifications.
2. A host specification consists of one or more host names, each prefixed by a colon (**:**). For example: **:indy :far.away.domain.com :localhost**
3. A missing host specification implies the default **pmie** source of metrics, as defined by a **-h** option on the command line, or the first named archive in an **-a** option on the command line, or PMCD on the local host.
4. An instance specification consists of one or more instance names, each prefixed by a hash or pound (**#**) sign. For example: **#eth0 #eth2**

   Recall that you can discover the instance names for a particular metric, using the pminfo command. See Section 5.2.1, "*pmie use of PCP services*".

Within the **pmie** grammar, an instance name is an identifier. If the instance name contains characters other than alphanumeric characters, enclose the instance name in single quotes; for example, **#\'/boot\' #\'/usr\'**

5. A missing instance specification implies all instances for the associated performance metric from each associated **pmie** source of metrics.
6. A sample time specification consists of either a single time or a range of times. A single time is represented as an at (@) followed by a natural number. A range of times is an at (@), followed by a natural number, followed by two periods (**..**) followed by a second natural number. The ordering of the end points in a range is immaterial. For example, **@0..9** specifies the last 10 sample times.
7. A missing sample time specification implies the most recent sample time.

The following metric expression refers to a three-dimensional set of values, with two hosts in one dimension, five sample times in another, and the number of instances in the third dimension being determined by the number of configured disk spindles on the two hosts.

```
disk.dev.read :foo :bar @0..4
```

7.3.4 pmie Rate Conversion

Many of the metrics delivered by PCP are cumulative counters. Consider the following metric:

```
disk.all.total
```

A single value for this metric tells you only that a certain number of disk I/O operations have occurred since boot time, and that information may be invalid if the counter has exceeded its 32-bit range and wrapped. You need at least two values, sampled at known times, to compute the recent rate at which the I/O operations are being executed. The required syntax would be this:

```
(disk.all.total @0 - disk.all.total @1) / delta
```

The accuracy of **delta** as a measure of actual inter-sample delay is an issue. **pmie** requests samples, at intervals of approximately **delta**, while the results exported from PMCD are time stamped with the high-resolution system clock time when the samples were extracted. For these reasons, a built-in and implicit rate conversion using accurate time stamps is provided by **pmie** for performance metrics that have counter semantics. For example, the following expression is unconditionally converted to a rate by pmie.

```
disk.all.total
```

7.3.5 pmie Arithmetic Expressions

Within **pmie**, simple arithmetic expressions are constructed from metrics expressions (see Section 5.3.3, "*pmie Metric Expressions*") and numeric constants, using all of the arithmetic operators and precedence rules of the C programming language.

All **pmie** arithmetic is performed in double precision.

Section 5.3.8, "*pmie Intrinsic Operators*", describes additional operators that may be used for aggregate operations to reduce the dimensionality of an arithmetic expression.

7.3.6 pmie Logical Expressions

A number of logical expression types are supported:

- Logical constants
- Relational expressions
- Boolean expressions
- Quantification operators

Logical Constants

Like in the C programming language, **pmie** interprets an arithmetic value of zero to be false, and all other arithmetic values are considered true.

Relational Expressions

Relational expressions are the simplest form of logical expression, in which values may be derived from arithmetic expressions using **pmie** relational operators. For example, the following is a relational expression that is true or false, depending on the aggregate total of disk read operations per second being greater than 50.

```
disk.all.read > 50 count/sec
```

All of the relational logical operators and precedence rules of the C programming language are supported in **pmie**.

As described in Section 5.3.3, "*pmie Metric Expressions*", arithmetic expressions in **pmie** may assume set values. The relational operators are also required to take constant, singleton, and set-valued expressions as arguments. The result has the same dimensionality as the operands. Suppose the rule in *Example 5.6. Relational Expressions* is given:

Example 5.6. Relational Expressions

```
hosts = ":gonzo";
intfs = "#eth0 #eth2";
all_intf = network.interface.in.packets
               $hosts $intfs @0..2 > 300 count/sec;
```

Then the execution of **pmie** may proceed as follows:

```
pmie -V uag.11
all_intf:
       gonzo: [eth0]      ?      ?      ?
       gonzo: [eth2]      ?      ?      ?
all_intf:
       gonzo: [eth0]  false      ?      ?
       gonzo: [eth2]  false      ?      ?
all_intf:
       gonzo: [eth0]   true  false      ?
       gonzo: [eth2]  false  false      ?
all_intf:
       gonzo: [eth0]   true   true  false
       gonzo: [eth2]  false  false  false
```

At each sample, the relational operator greater than (>) produces six truth values for the cross-product of the **instance** and **sample time** dimensions.

Section 5.3.6.4, "*Quantification Operators*", describes additional logical operators that may be used to reduce the dimensionality of a relational expression.

Boolean Expressions

The regular Boolean operators from the C programming language are supported: conjunction (**&&**), disjunction (||) and negation (**!**).

As with the relational operators, the Boolean operators accommodate set-valued operands, and set-valued results.

Quantification Operators

Boolean and relational operators may accept set-valued operands and produce set-valued results. In many cases, rules that are appropriate for performance management require a set of truth values to be reduced along one or more of the dimensions of hosts, instances, and sample times described in Section 5.3.3, "*pmie Metric Expressions*". The **pmie** quantification operators perform this function.

Each quantification operator takes a one-, two-, or three-dimension set of truth values as an operand, and reduces it to a set of smaller dimension, by quantification along a single dimension. For example, suppose the expression in the previous example is simplified and prefixed by **some_sample**, to produce the following expression:

```
intfs = "#eth0 #eth2";
all_intf = some_sample network.interface.in.packets
                    $intfs @0..2 > 300 count/sec;
```

Then the expression result is reduced from six values to two (one per interface instance), such that the result for a particular instance will be false unless the relational expression for the same interface instance is true for at least one of the preceding three sample times.

There are existential, universal, and percentile quantification operators in each of the *host, instance*, and *sample time* dimensions to produce the nine operators as follows:

some_host	True if the expression is true for at least one host for the same instance and sample time.
all_host	True if the expression is true for every host for the same instance and sample time.
N%_host	True if the expression is true for at least N% of the hosts for the same instance and sample time.
some_inst	True if the expression is true for at least one instance for the same host and sample time.
all_instance	True if the expression is true for every instance for the same host and sample time.
N%_instance	True if the expression is true for at least N% of the instances for the same host and sample time.
some_sample time	True if the expression is true for at least one sample time for the same host and instance.
all_sample time	True if the expression is true for every sample time for the same host and instance.
N%_sample time	True if the expression is true for at least N% of the sample times for the same host and instance.

These operators may be nested. For example, the following expression answers the question: "Are all hosts experiencing at least 20% of their disks busy either reading or writing?"

```
Servers = ":moomba :babylon";
all_host (
    20%_inst disk.dev.read $Servers > 40 ||
    20%_inst disk.dev.write $Servers > 40
);
```

The following expression uses different syntax to encode the same semantics:

```
all_host (
    20%_inst (
        disk.dev.read $Servers > 40 ||
        disk.dev.write $Servers > 40
    )
);
```

Note: To avoid confusion over precedence and scope for the quantification operators, use explicit parentheses.

Two additional quantification operators are available for the instance dimension only, namely **match_inst** and **nomatch_inst**, that take a regular expression and a boolean expression. The result is the boolean AND of the expression and the result of matching (or not matching) the associated instance name against the regular expression.

For example, this rule evaluates error rates on various 10BaseT Ethernet network interfaces (such as ecN, ethN, or efN):

```
some_inst
        match_inst "^(ec|eth|ef)"
                network.interface.total.errors > 10 count/sec
-> syslog "Ethernet errors:" " %i"
```

7.3.7 pmie Rule Expressions

Rule expressions for **pmie** have the following syntax:

```
lexpr -> actions ;
```

The semantics are as follows:

- If the logical expression **lexpr** evaluates **true**, then perform the *actions* that follow. Otherwise, do not perform the *actions*.
- It is required that **lexpr** has a singular truth value. Aggregation and quantification operators must have been applied to reduce multiple truth values to a single value.
- When executed, an *action* completes with a success/failure status.
- One or more *actions* may appear; consecutive *actions* are separated by operators that control the execution of subsequent *actions*, as follows:
  - *action-1* **&** : Always execute subsequent actions (serial execution).
  - *action-1* **|** : If *action-1* fails, execute subsequent actions, otherwise skip the subsequent actions (alternation).

An *action* is composed of a keyword to identify the action method, an optional *time* specification, and one or more arguments.

A *time* specification uses the same syntax as a valid time interval that may be assigned to **delta**, as described in Section 5.3.2, "*Setting Evaluation Frequency* ". If the *action* is executed and the *time* specification is present, **pmie** will suppress any subsequent execution of this *action* until the wall clock time has advanced by *time*.

The arguments are passed directly to the action method.

The following action methods are provided:

shell

The single argument is passed to the shell for execution. This *action* is implemented using **system** in the background. The *action* does not wait for the system call to return, and succeeds unless the fork fails.

alarm

A notifier containing a time stamp, a single *argument* as a message, and a **Cancel** button is posted on the current display screen (as identified by the **DISPLAY** environment variable). Each alarm *action* first checks if its notifier is already active. If there is an identical active notifier, a duplicate notifier is not posted. The action succeeds unless the fork fails.

syslog

A message is written into the system log. If the first word of the first argument is **-p**, the second word is interpreted as the priority (see the **syslog(3)** man page); the message tag is **pcp-pmie**. The remaining argument is the message to be written to the system log. This action always succeeds.

print

A message containing a time stamp in **ctime(3)** format and the argument is displayed out to standard output (**stdout**). This action always succeeds.

Within the argument passed to an action method, the following expansions are supported to allow some of the context from the logical expression on the left to appear to be embedded in the argument:

%h	The value of a *host* that makes the expression true.
%i	The value of an *instance* that makes the expression true.
%v	The value of a performance metric from the logical expression.

Some ambiguity may occur in respect to which host, instance, or performance metric is bound to a %-token. In most cases, the leftmost binding in the top-level subexpression is used. You may need to use **pmie** in the interactive debugging mode (specify the **-d** command line option) in conjunction with the **-W** command line option to discover which subexpressions contributes to the %-token bindings.

Example 5.7. Rule Expression Options illustrates some of the options when constructing rule expressions:

Example 5.7. Rule Expression Options

```
some_inst ( disk.dev.total > 60 )
       -> syslog 10 mins "[%i] busy, %v IOPS " &
          shell 1 hour "echo \
               'Disk %i is REALLY busy. Running at %v I/Os per second' \
               | Mail -s 'pmie alarm' sysadm";
```

In this case, **%v** and **%i** are both associated with the instances for the metric **disk.dev.total** that make the expression true. If more than one instance makes the expression true (more than one disk is busy), then the argument is formed by concatenating the result from each %-token binding. The text added to the system log file might be as shown in *Example 5.8. System Log Text* :

Example 5.8. System Log Text

```
Aug 6 08:12:44 5B:gonzo pcp-pmie[3371]:
                          [disk1] busy, 3.7 IOPS [disk2] busy, 0.3 IOPS
```

Note: When **pmie** is processing performance metrics from a set of PCP archive logs, the *actions* will be processed in the expected manner; however, the action methods are modified to report a textual facsimile of the *action* on the standard output.

Consider the rule in *Example 5.9. Standard Output* :

Example 5.9. Standard Output

```
delta = 2 sec;  // more often for demonstration purposes
percpu  = "kernel.percpu";
// Unusual usr-sys split when some CPU is more than 20% in usr mode
// and sys mode is at least 1.5 times usr mode
//
cpu_usr_sys = some_inst (
        $percpu.cpu.sys > $percpu.cpu.user * 1.5 &&
        $percpu.cpu.user > 0.2
   ) ->  alarm "Unusual sys time: " "%i ";
```

When evaluated against an archive, the following output is generated (the alarm action produces a message on standard output):

```
pmafm ${HOME}/f4 pmie cpu.head cpu.00
alarm Wed Aug  7 14:54:48 2012: Unusual sys time: cpu0
alarm Wed Aug  7 14:54:50 2012: Unusual sys time: cpu0
alarm Wed Aug  7 14:54:52 2012: Unusual sys time: cpu0
alarm Wed Aug  7 14:55:02 2012: Unusual sys time: cpu0
alarm Wed Aug  7 14:55:06 2012: Unusual sys time: cpu0
```

7.3.8 pmie Intrinsic Operators

The following sections describe some other useful intrinsic operators for **pmie**. These operators are divided into three groups:

1. Arithmetic aggregation
2. The rate operator
3. Transitional operators

Arithmetic Aggregation

For set-valued arithmetic expressions, the following operators reduce the dimensionality of the result by arithmetic aggregation along one of the *host*, *instance*, or *sample time* dimensions. For example, to aggregate in the *host* dimension, the following operators are provided:

avg_host	Computes the average value across all *instances* for the same *host* and *sample time*
sum_host	Computes the total value across all *instances* for the same *host* and *sample time*
count_host	Computes the number of values across all *instances* for the same *host* and *sample time*
min_host	Computes the minimum value across all *instances* for the same *host* and *sample time*
max_host	Computes the maximum value across all *instances* for the same *host* and *sample time*

Ten additional operators correspond to the forms *_inst and *_sample.

The following example illustrates the use of an aggregate operator in combination with an existential operator to answer the question “Does some host currently have two or more busy processors?”

```
// note '' to escape - in host name
poke = ":moomba :'mac-larry' :bitbucket";
some_host (
    count_inst ( kernel.percpu.cpu.user $poke +
                 kernel.percpu.cpu.sys $poke > 0.7 ) >= 2
```

(continues on next page)

(continued from previous page)

```
    )
        -> alarm "2 or more busy CPUs";
```

The rate Operator

The **rate** operator computes the rate of change of an arithmetic expression as shown in the following example:

```
rate mem.util.cached
```

It returns the rate of change for the **mem.util.cached** performance metric; that is, the rate at which page cache memory is being allocated and released.

The **rate** intrinsic operator is most useful for metrics with instantaneous value semantics. For metrics with counter semantics, **pmie** already performs an implicit rate calculation (see the Section 5.3.4, "*pmie Rate Conversion*") and the **rate** operator would produce the second derivative with respect to time, which is less likely to be useful.

Transitional Operators

In some cases, an action needs to be triggered when an expression changes from true to false or vice versa. The following operators take a logical expression as an operand, and return a logical expression:

- **rising**: Has the value **true** when the operand transitions from **false** to **true** in consecutive samples.
- **falling**: Has the value **false** when the operand transitions from **true** to **false** in consecutive samples.

7.4 pmie Examples

The examples presented in this section are task-oriented and use the full power of the pmie specification language as described in Section 5.3, "*Specification Language for pmie*".

Source code for the **pmie** examples in this chapter, and many more examples, is provided within the *PCP Tutorials and Case Studies*. *Example 5.10. Monitoring CPU Utilization* and *Example 5.11. Monitoring Disk Activity* illustrate monitoring CPU utilization and disk activity.

Example 5.10. Monitoring CPU Utilization

```
// Some Common Performance Monitoring Scenarios
//
// The CPU Group
//
delta = 2 sec;  // more often for demonstration purposes
// common prefixes
//
percpu  = "kernel.percpu";
all     = "kernel.all";
// Unusual usr-sys split when some CPU is more than 20% in usr mode
// and sys mode is at least 1.5 times usr mode
//
cpu_usr_sys =
       some_inst (
           $percpu.cpu.sys > $percpu.cpu.user * 1.5 &&
           $percpu.cpu.user > 0.2
       )
```

(continues on next page)

(continued from previous page)

```
            ->  alarm "Unusual sys time: " "%i ";
// Over all CPUs, syscall_rate > 1000 * no_of_cpus
//
cpu_syscall =
       $all.syscall > 1000 count/sec * hinv.ncpu
       ->  print "high aggregate syscalls: %v";
// Sustained high syscall rate on a single CPU
//
delta = 30 sec;
percpu_syscall =
       some_inst (
           $percpu.syscall > 2000 count/sec
       )
           -> syslog "Sustained syscalls per second? " "[%i] %v ";
// the 1 minute load average exceeds 5 * number of CPUs on any host
hosts = ":gonzo :moomba";   // change as required
delta = 1 minute;           // no need to evaluate more often than this
high_load =
     some_host (
           $all.load $hosts #'1 minute' > 5 * hinv.ncpu
       )
           -> alarm "High Load Average? " "%h: %v ";
```

Example 5.11. Monitoring Disk Activity

```
// Some Common Performance Monitoring Scenarios
//
// The Disk Group
//
delta = 15 sec;         // often enough for disks?
// common prefixes
//
disk    = "disk";
// Any disk performing more than 40 I/Os per second, sustained over
// at least 30 seconds is probably busy
//
delta = 30 seconds;
disk_busy =
       some_inst (
           $disk.dev.total > 40 count/sec
       )
]      -> shell "Mail -s 'Heavy sustained disk traffic' sysadm";
// Try and catch bursts of activity ... more than 60 I/Os per second
// for at least 25% of 8 consecutive 3 second samples
//
delta = 3 sec;
disk_burst =
       some_inst (
           25%_sample (
               $disk.dev.total @0..7 > 60 count/sec
           )
       )
       -> alarm "Disk Burst? " "%i ";
// any SCSI disk controller performing more than 3 Mbytes per
// second is busy
// Note: the obscure 512 is to convert blocks/sec to byte/sec,
//       and pmie handles the rest of the scale conversion
```

(continues on next page)

(continued from previous page)

```
//
some_inst $disk.ctl.blktotal * 512 > 3 Mbyte/sec
          -> alarm "Busy Disk Controller: " "%i ";
```

7.5 Developing and Debugging pmie Rules

Given the **-d** command line option, **pmie** executes in interactive mode, and the user is presented with a menu of options:

```
pmie debugger commands
     f [file-name]      - load expressions from given file or stdin
     l [expr-name]      - list named expression or all expressions
     r [interval]       - run for given or default interval
     S time-spec        - set start time for run
     T time-spec        - set default interval for run command
     v [expr-name]      - print subexpression for %h, %i and %v bindings
     h or ?             - print this menu of commands
     q                  - quit
pmie>
```

If both the **-d** option and a filename are present, the expressions in the given file are loaded before entering interactive mode. Interactive mode is useful for debugging new rules.

7.6 Caveats and Notes on pmie

The following sections provide important information for users of **pmie**.

7.6.1 Performance Metrics Wraparound

Performance metrics that are cumulative counters may occasionally overflow their range and wraparound to 0. When this happens, an unknown value (printed as **?**) is returned as the value of the metric for one sample (recall that the value returned is normally a rate). You can have PCP interpolate a value based on expected rate of change by setting the **PCP_COUNTER_WRAP** environment variable.

7.6.2 pmie Sample Intervals

The sample interval (**delta**) should always be long enough, particularly in the case of rates, to ensure that a meaningful value is computed. Interval may vary according to the metric and your needs. A reasonable minimum is in the range of ten seconds or several minutes. Although PCP supports sampling rates up to hundreds of times per second, using small sample intervals creates unnecessary load on the monitored system.

7.6.3 pmie Instance Names

When you specify a metric instance name (*#identifier*) in a **pmie** expression, it is compared against the instance name looked up from either a live collector system or an archive as follows:

- If the given instance name and the looked up name are the same, they are considered to match.
- Otherwise, the first two space separated tokens are extracted from the looked up name. If the given instance name is the same as either of these tokens, they are considered a match.

For some metrics, notably the per process (**proc.xxx.xxx**) metrics, the first token in the looked up instance name is impossible to determine at the time you are writing **pmie** expressions. The above policy circumvents this problem.

7.6.4 pmie Error Detection

The parser used in **pmie** is not particularly robust in handling syntax errors. It is suggested that you check any problematic expressions individually in interactive mode:

```
pmie -v -d
pmie> f
expression
Ctrl+D
```

If the expression was parsed, its internal representation is shown:

```
pmie> l
```

The expression is evaluated twice and its value printed:

```
pmie> r 10sec
```

Then quit:

```
pmie> q
```

It is not always possible to detect semantic errors at parse time. This happens when a performance metric descriptor is not available from the named host at this time. A warning is issued, and the expression is put on a wait list. The wait list is checked periodically (about every five minutes) to see if the metric descriptor has become available. If an error is detected at this time, a message is printed to the standard error stream (**stderr**) and the offending expression is set aside.

7.7 Creating pmie Rules with pmieconf

The **pmieconf** tool is a command line utility that is designed to aid the specification of **pmie** rules from parameterized versions of the rules. **pmieconf** is used to display and modify variables or parameters controlling the details of the generated **pmie** rules.

pmieconf reads two different forms of supplied input files and produces a localized **pmie** configuration file as its output.

The first input form is a generalized **pmie** rule file such as those found below `${PCP_VAR_DIR}/config/pmieconf`. These files contain the generalized rules which **pmieconf** is able to manipulate. Each of the rules can be enabled or disabled, or the individual variables associated with each rule can be edited.

The second form is an actual **pmie** configuration file (that is, a file which can be interpreted by **pmie**, conforming to the **pmie** syntax described in Section 5.3, "*Specification Language for pmie*"). This file is both input to and output from **pmieconf**.

The input version of the file contains any changed variables or rule states from previous invocations of **pmieconf**, and the output version contains both the changes in state (for any subsequent **pmieconf** sessions) and the generated **pmie** syntax. The **pmieconf** state is embedded within a **pmie** comment block at the head of the output file and is not interpreted by **pmie** itself.

pmieconf is an integral part of the **pmie** daemon management process described in Section 5.8, "*Management of pmie Processes*". *Procedure 5.1. Display pmieconf Rules* and *Procedure 5.2. Modify pmieconf Rules and Generate a pmie File* introduce the **pmieconf** tool through a series of typical operations.

Procedure 5.1. Display pmieconf Rules

1. Start **pmieconf** interactively (as the superuser).

```
pmieconf -f ${PCP_SYSCONF_DIR}/pmie/config.demo
Updates will be made to ${PCP_SYSCONF_DIR}/pmie/config.demo

pmieconf>
```

2. List the set of available **pmieconf** rules by using the **rules** command.
3. List the set of rule groups using the **groups** command.
4. List only the enabled rules, using the **rules enabled** command.
5. List a single rule:

```
pmieconf> list memory.swap_low
   rule: memory.swap_low  [Low free swap space]
   help: There is only threshold percent swap space remaining - the system
         may soon run out of virtual memory.  Reduce the number and size␣
↪of
         the running programs or add more swap(1) space before it
completely
         runs out.
         predicate =
           some_host (
               ( 100 * ( swap.free $hosts$ / swap.length $hosts$ ) )
                 < $threshold$
               && swap.length $hosts$ > 0        // ensure swap in use
            )
   vars: enabled = no
         threshold = 10%

pmieconf>
```

6. List one rule variable:

```
pmieconf> list memory.swap_low threshold
   rule: memory.swap_low  [Low free swap space]
         threshold = 10%

pmieconf>
```

Procedure 5.2. Modify pmieconf Rules and Generate a pmie File

1. Lower the threshold for the **memory.swap_low** rule, and also change the **pmie** sample interval affecting just this rule. The **delta** variable is special in that it is not associated with any particular rule; it has been defined as

a global **pmieconf** variable. Global variables can be displayed using the **list global** command to **pmieconf**, and can be modified either globally or local to a specific rule.

```
pmieconf> modify memory.swap_low threshold 5

pmieconf> modify memory.swap_low delta "1 sec"

pmieconf>
```

2. Disable all of the rules except for the **memory.swap_low** rule so that you can see the effects of your change in isolation.

   This produces a relatively simple **pmie** configuration file:

```
pmieconf> disable all

pmieconf> enable memory.swap_low

pmieconf> status
  verbose:  off
  enabled rules:  1 of 35
  pmie configuration file:  ${PCP_SYSCONF_DIR}/pmie/config.demo
  pmie processes (PIDs) using this file:  (none found)

pmieconf> quit
```

   You can also use the **status** command to verify that only one rule is enabled at the end of this step.

3. Run **pmie** with the new configuration file. Use a text editor to view the newly generated **pmie** configuration file (`${PCP_SYSCONF_DIR}/pmie/config.demo`), and then run the command:

```
pmie -T "1.5 sec" -v -l ${HOME}/demo.log ${PCP_SYSCONF_DIR}/pmie/config.demo
memory.swap_low: false

memory.swap_low: false

cat ${HOME}/demo.log
Log for pmie on venus started Mon Jun 21 16:26:06 2012

pmie: PID = 21847, default host = venus

[Mon Jun 21 16:26:07] pmie(21847) Info: evaluator exiting

Log finished Mon Jun 21 16:26:07 2012
```

4. Notice that both of the **pmieconf** files used in the previous step are simple text files, as described in the **pmieconf(5)** man page:

```
file ${PCP_SYSCONF_DIR}/pmie/config.demo
${PCP_SYSCONF_DIR}/pmie/config.demo:  PCP pmie config (V.1)
file ${PCP_VAR_DIR}/config/pmieconf/memory/swap_low
${PCP_VAR_DIR}/config/pmieconf/memory/swap_low:       PCP pmieconf rules (V.1)
```

7.8 Management of pmie Processes

The **pmie** process can be run as a daemon as part of the system startup sequence, and can thus be used to perform automated, live performance monitoring of a running system. To do this, run these commands (as superuser):

```
chkconfig pmie on
${PCP_RC_DIR}/pmie start
```

By default, these enable a single **pmie** process monitoring the local host, with the default set of **pmieconf** rules enabled (for more information about **pmieconf**, see Section 5.7, "*Creating pmie Rules with pmieconf*"). *Procedure 5.3. Add a New pmie Instance to the pmie Daemon Management Framework* illustrates how you can use these commands to start any number of **pmie** processes to monitor local or remote machines.

Procedure 5.3. Add a New pmie Instance to the pmie Daemon Management Framework

1. Use a text editor (as superuser) to edit the `pmie${PCP_PMIECONTROL_PATH}` and `${PCP_PMIECONTROL_PATH}.d` control files. Notice the default entry, which looks like this:

   ```
   #Host           P?  S?  Log File                                   Arguments
   LOCALHOSTNAME   y   n   PCP_LOG_DIR/pmie/LOCALHOSTNAME/pmie.log   -c config.
   ↪default
   ```

   This entry is used to enable a local **pmie** process. Add a new entry for a remote host on your local network (for example, **venus**), by using your pmie configuration file (see Section 5.7, "*Creating pmie Rules with pmieconf*"):

   ```
   #Host           P?  S?  Log File                                   Arguments
   venus           n   n   PCP_LOG_DIR/pmie/venus/pmie.log          -c config.demo
   ```

   Note: Without an absolute path, the configuration file (**-c** above) will be resolved using `${PCP_SYSCONF_DIR}/pmie` - if **config.demo** was created in *Procedure 5.2. Modify pmieconf Rules and Generate a pmie File* it would be used here for host **venus**, otherwise a new configuration file will be generated using the default rules (at `${PCP_SYSCONF_DIR}/pmie/config.demo`).

2. Enable **pmie** daemon management:

   ```
   chkconfig pmie on
   ```

This simple step allows **pmie** to be started as part of your machine's boot process.

3. Start the two **pmie** daemons. At the end of this step, you should see two new **pmie** processes monitoring the local and remote hosts:

   ```
   ${PCP_RC_DIR}/pmie start
   Performance Co-Pilot starting inference engine(s) ...
   ```

Wait a few moments while the startup scripts run. The **pmie** start script uses the **pmie_check** script to do most of its work.

Verify that the **pmie** processes have started:

```
pcp
Performance Co-Pilot configuration on pluto:

 platform: Linux pluto 3.10.0-0.rc7.64.el7.x86_64 #1 SMP
 hardware: 8 cpus, 2 disks, 23960MB RAM
```

(continues on next page)

(continued from previous page)

```
timezone: EST-10
    pmcd: Version 3.11.3-1, 8 agents
    pmda: pmcd proc xfs linux mmv infiniband gluster elasticsearch
    pmie: pluto: ${PCP_LOG_DIR}/pmie/pluto/pmie.log
          venus: ${PCP_LOG_DIR}/pmie/venus/pmie.log
```

If a remote host is not up at the time when **pmie** is started, the **pmie** process may exit. **pmie** processes may also exit if the local machine is starved of memory resources. To counter these adverse cases, it can be useful to have a **crontab** entry running. Adding an entry as shown in Section 5.8.1, "*Add a pmie crontab Entry*" ensures that if one of the configured **pmie** processes exits, it is automatically restarted.

Note: Depending on your platform, the **crontab** entry discussed here may already have been installed for you, as part of the package installation process. In this case, the file **/etc/cron.d/pcp-pmie** will exist, and the rest of this section can be skipped.

7.8.1 Add a pmie crontab Entry

To activate the maintenance and housekeeping scripts for a collection of inference engines, execute the following tasks while logged into the local host as the superuser (**root**):

1. Augment the **crontab** file for the **pcp** user. For example:

   ```
   crontab -l -u pcp > ${HOME}/crontab.txt
   ```

2. Edit `${HOME}/crontab.txt`, adding lines similar to those from the sample `${PCP_VAR_DIR}/config/pmie/crontab` file for **pmie_daily** and **pmie_check**; for example:

   ```
   # daily processing of pmie logs
   10     0     *     *     *    ${PCP_BINADM_DIR}/pmie_daily
   # every 30 minutes, check pmie instances are running
   25,55  *     *     *     *    ${PCP_BINADM_DIR}/pmie_check
   ```

3. Make these changes permanent with this command:

   ```
   crontab -u pcp < ${HOME}/crontab.txt
   ```

7.8.2 Global Files and Directories

The following global files and directories influence the behavior of **pmie** and the **pmie** management scripts:

`${PCP_DEMOS_DIR}/pmie/*`

Contains sample **pmie** rules that may be used as a basis for developing local rules.

`${PCP_SYSCONF_DIR}/pmie/config.default`

Is the default **pmie** configuration file that is used when the **pmie** daemon facility is enabled. Generated by **pmieconf** if not manually setup beforehand.

`${PCP_VAR_DIR}/config/pmieconf/*/*`

Contains the **pmieconf** rule definitions (templates) in its subdirectories.

`${PCP_PMIECONTROL_PATH}` and `${PCP_PMIECONTROL_PATH}.d` files

Defines which PCP collector hosts require a daemon **pmie** to be launched on the local host, where the configuration file comes from, where the **pmie** log file should be created, and **pmie** startup options.

`${PCP_VAR_DIR}/config/pmlogger/crontab`

Contains default **crontab** entries that may be merged with the **crontab** entries for root to schedule the periodic execution of the **pmie_check** script, for verifying that **pmie** instances are running. Only for platforms where a default **crontab** is not automatically installed during the initial PCP package installation.

`${PCP_LOG_DIR}/pmie/*`

Contains the **pmie** log files for the host. These files are created by the default behavior of the `${PCP_RC_DIR}/pmie` startup scripts.

7.8.3 pmie Instances and Their Progress

The PMCD PMDA exports information about executing **pmie** instances and their progress in terms of rule evaluations and action execution rates.

`pmie_check`

This command is similar to the **pmlogger** support script, **pmlogger_check**.

`${PCP_RC_DIR}/pmie`

This start script supports the starting and stopping of multiple **pmie** instances that are monitoring one or more hosts.

`${PCP_TMP_DIR}/pmie`

The statistics that **pmie** gathers are maintained in binary data structure files. These files are located in this directory.

`pmcd.pmie metrics`

If **pmie** is running on a system with a PCP collector deployment, the PMCD PMDA exports these metrics via the **pmcd.pmie** group of metrics.

CHAPTER

EIGHT

ARCHIVE LOGGING

Contents

- *Archive Logging*
  - *Introduction to Archive Logging*
    - *Archive Logs and the PMAPI*
    - *Retrospective Analysis Using Archive Logs*
    - *Using Archive Logs for Capacity Planning*
  - *Using Archive Logs with Performance Tools*
    - *Coordination between pmlogger and PCP tools*
    - *Administering PCP Archive Logs Using cron Scripts*
    - *Archive Log File Management*
      - *Basename Conventions*
      - *Log Volumes*
      - *Basenames for Managed Archive Log Files*
      - *Directory Organization for Archive Log Files*
      - *Configuration of pmlogger*
      - *PCP Archive Contents*
  - *Cookbook for Archive Logging*
    - *Primary Logger*
    - *Other Logger Configurations*
    - *Archive Log Administration*
  - *Other Archive Logging Features and Services*
    - *PCP Archive Folios*
    - *Manipulating Archive Logs with pmlogextract*
    - *Summarizing Archive Logs with pmlogsummary*
    - *Primary Logger*
    - *Using pmlc*

- *Archive Logging Troubleshooting*
  - *pmlogger Cannot Write Log*
  - *Cannot Find Log*
  - *Primary pmlogger Cannot Start*
  - *Identifying an Active pmlogger Process*
  - *Illegal Label Record*
  - *Empty Archive Log Files or pmlogger Exits Immediately*

Performance monitoring and management in complex systems demands the ability to accurately capture performance characteristics for subsequent review, analysis, and comparison. Performance Co-Pilot (PCP) provides extensive support for the creation and management of archive logs that capture a user-specified profile of performance information to support retrospective performance analysis.

The following major sections are included in this chapter:

Section 6.1, "*Introduction to Archive Logging*", presents the concepts and issues involved with creating and using archive logs.

Section 6.2, "*Using Archive Logs with Performance Tools*", describes the interaction of the PCP tools with archive logs.

Section 6.3, "*Cookbook for Archive Logging*", shows some shortcuts for setting up useful PCP archive logs.

Section 6.4, "*Other Archive Logging Features and Services*", provides information about other archive logging features and services.

Section 6.5, "*Archive Logging Troubleshooting*", presents helpful directions if your archive logging implementation is not functioning correctly.

8.1 Introduction to Archive Logging

Within the PCP, the **pmlogger** utility may be configured to collect archives of performance metrics. The archive creation process is simple and very flexible, incorporating the following features:

- Archive log creation at either a PCP collector (typically a server) or a PCP monitor system (typically a workstation), or at some designated PCP archive logger host.
- Concurrent independent logging, both local and remote. The performance analyst can activate a private **pmlogger** instance to collect only the metrics of interest for the problem at hand, independent of other logging on the workstation or remote host.
- Independent determination of logging frequency for individual metrics or metric instances. For example, you could log the "5 minute" load average every half hour, the write I/O rate on the DBMS log spindle every 10 seconds, and aggregate I/O rates on the other disks every minute.
- Dynamic adjustment of what is to be logged, and how frequently, via **pmlc**. This feature may be used to disable logging or to increase the sample interval during periods of low activity or chronic high activity. A local **pmlc** may interrogate and control a remote **pmlogger**, subject to the access control restrictions implemented by **pmlogger**.
- Self-contained logs that include all system configuration and metadata required to interpret the values in the log. These logs can be kept for analysis at a much later time, potentially after the hardware or software has been reconfigured and the logs have been stored as discrete, autonomous files for remote analysis. The logs are endian-neutral and platform independent - there is no requirement that the monitor host machine used for

analysis be similar to the collector machine in any way, nor do they have to have the same versions of PCP. PCP archives created over 15 years ago can still be replayed with the current versions of PCP!

- **cron**-based scripts to expedite the operational management, for example, log rotation, consolidation, and culling. Another helper tool, **pmlogconf** can be used to generate suitable logging configurations for a variety of situations.
- Archive folios as a convenient aggregation of multiple archive logs. Archive folios may be created with the **mkaf** utility and processed with the **pmafm** tool.

8.1.1 Archive Logs and the PMAPI

Critical to the success of the PCP archive logging scheme is the fact that the library routines providing access to real-time feeds of performance metrics also provide access to the archive logs.

Live feeds (or real-time) sources of performance metrics and archives are literally interchangeable, with a single Performance Metrics Application Programming Interface (PMAPI) that preserves the same semantics for both styles of metric source. In this way, applications and tools developed against the PMAPI can automatically process either live or historical performance data.

8.1.2 Retrospective Analysis Using Archive Logs

One of the most important applications of archive logging services provided by PCP is in the area of retrospective analysis. In many cases, understanding today's performance problems can be assisted by side-by-side comparisons with yesterday's performance. With routine creation of performance archive logs, you can concurrently replay pictures of system performance for two or more periods in the past.

Archive logs are also an invaluable source of intelligence when trying to diagnose what went wrong, as in a performance post-mortem. Because the PCP archive logs are entirely self-contained, this analysis can be performed off-site if necessary.

Each archive log contains metric values from only one host. However, many PCP tools can simultaneously visualize values from multiple archives collected from different hosts.

The archives can be replayed using the inference engine (**pmie** is an application that uses the PMAPI). This allows you to automate the regular, first-level analysis of system performance.

Such analysis can be performed by constructing suitable expressions to capture the essence of common resource saturation problems, then periodically creating an archive and playing it against the expressions. For example, you may wish to create a daily performance audit (perhaps run by the cron command) to detect performance regressions.

For more about pmie, see Chapter 5, *Performance Metrics Inference Engine*.

8.1.3 Using Archive Logs for Capacity Planning

By collecting performance archives with relatively long sampling periods, or by reducing the daily archives to produce summary logs, the capacity planner can collect the base data required for forward projections, and can estimate resource demands and explore "what if" scenarios by replaying data using visualization tools and the inference engine.

8.2 Using Archive Logs with Performance Tools

Most PCP tools default to real-time display of current values for performance metrics from PCP collector host(s). However, most PCP tools also have the capability to display values for performance metrics retrieved from PCP archive log(s). The following sections describe plans, steps, and general issues involving archive logs and the PCP tools.

8.2.1 Coordination between pmlogger and PCP tools

Most commonly, a PCP tool would be invoked with the **-a** option to process sets of archive logs some time after pmlogger had finished creating the archive. However, a tool such as **pmchart** that uses a Time Control dialog (see Section 3.3, "*Time Duration and Control*") stops when the end of a set of archives is reached, but could resume if more data is written to the PCP archive log.

Note: **pmlogger** uses buffered I/O to write the archive log so that the end of the archive may be aligned with an I/O buffer boundary, rather than with a logical archive log record. If such an archive was read by a PCP tool, it would appear truncated and might confuse the tool. These problems may be avoided by sending **pmlogger** a **SIGUSR1** signal, or by using the **flush** command of **pmlc** to force **pmlogger** to flush its output buffers.

8.2.2 Administering PCP Archive Logs Using cron Scripts

Many operating systems support the **cron** process scheduling system.

PCP supplies shell scripts to use the **cron** functionality to help manage your archive logs. The following scripts are supplied:

Script	Description
pmlogger_daily(1)	Performs a daily housecleaning of archive logs and notices.
pmlogger_merge(1)	Merges archive logs and is called by **pmlogger_daily**.
pmlogger_check(1)	Checks to see that all desired **pmlogger** processes are running on your system, and invokes any that are missing for any reason.
pmlogconf(1)	Generates suitable **pmlogger** configuration files based on a pre-defined set of templates. It can probe the state of the system under observation to make informed decisions about which metrics to record. This is an extensible facility, allowing software upgrades and new PMDA installations to add to the existing set of templates.
pmsnap(1)	Generates graphic image snapshots of **pmchart** performance charts at regular intervals.

The configuration files used by these scripts can be edited to suit your particular needs, and are generally controlled by the `${PCP_PMLOGGERCONTROL_PATH}` and `${PCP_PMLOGGERCONTROL_PATH}.d` files (**pmsnap** has an additional control file, `${PCP_PMSNAPCONTROL_PATH}`). Complete information on these scripts is available in the **pmlogger_daily(1)** and **pmsnap(1)** man pages.

8.2.3 Archive Log File Management

PCP archive log files can occupy a great deal of disk space, and management of archive logs can be a large task in itself. The following sections provide information to assist you in PCP archive log file management.

Basename Conventions

When a PCP archive is created by **pmlogger**, an archive basename must be specified and several physical files are created, as shown in *Table 6.1. Filenames for PCP Archive Log Components (archive.*)*.

Table 6.1. Filenames for PCP Archive Log Components (archive.*)

Filename	Contents
archive. *index*	Temporal index for rapid access to archive contents.
archive. *meta*	Metadata descriptions for performance metrics and instance domains appearing in the archive.
archive.N	Volumes of performance metrics values, for **N** = 0,1,2,. . .

Log Volumes

A single PCP archive may be partitioned into a number of volumes. These volumes may expedite management of the archive; however, the metadata file and at least one volume must be present before a PCP tool can process the archive.

You can control the size of an archive log volume by using the **-v** command line option to **pmlogger**. This option specifies how large a volume should become before pmlogger starts a new volume. Archive log volumes retain the same base filename as other files in the archive log, and are differentiated by a numeric suffix that is incremented with each volume change. For example, you might have a log volume sequence that looks like this:

```
netserver-log.0
netserver-log.1
netserver-log.2
```

You can also cause an existing log to be closed and a new one to be opened by sending a **SIGHUP** signal to **pmlogger**, or by using the **pmlc** command to change the **pmlogger** instructions dynamically, without interrupting **pmlogger** operation. Complete information on log volumes is found in the **pmlogger(1)** man page.

Basenames for Managed Archive Log Files

The PCP archive management tools support a consistent scheme for selecting the basenames for the files in a collection of archives and for mapping these files to a suitable directory hierarchy.

Once configured, the PCP tools that manage archive logs employ a consistent scheme for selecting the basename for an archive each time **pmlogger** is launched, namely the current date and time in the format YYYYMMDD.HH.MM. Typically, at the end of each day, all archives for a particular host on that day would be merged to produce a single archive with a basename constructed from the date, namely YYYYMMDD. The **pmlogger_daily** script performs this action and a number of other routine housekeeping chores.

Directory Organization for Archive Log Files

If you are using a deployment of PCP tools and daemons to collect metrics from a variety of hosts and storing them all at a central location, you should develop an organized strategy for storing and naming your log files.

Note: There are many possible configurations of **pmlogger**, as described in Section 7.3, "*PCP Archive Logger Deployment*". The directory organization described in this section is recommended for any system on which **pmlogger** is configured for permanent execution (as opposed to short-term executions, for example, as launched from **pmchart** to record some performance data of current interest).

Typically, the filesystem structure can be used to reflect the number of hosts for which a **pmlogger** instance is expected to be running locally, obviating the need for lengthy and cumbersome filenames. It makes considerable sense to place all logs for a particular host in a separate directory named after that host. Because each instance of **pmlogger** can only log metrics fetched from a single host, this also simplifies some of the archive log management and administration tasks.

For example, consider the filesystem and naming structure shown in *Figure 6.1. Archive Log Directory Structure*.

Fig. 1: Figure 6.1. Archive Log Directory Structure

The specification of where to place the archive log files for particular **pmlogger** instances is encoded in the `${PCP_PMLOGGERCONTROL_PATH}` and `${PCP_PMLOGGERCONTROL_PATH}.d` configuration files, and these files should be customized on each host running an instance of **pmlogger**.

If many archives are being created, and the associated PCP collector systems form peer classes based upon service type (Web servers, DBMS servers, NFS servers, and so on), then it may be appropriate to introduce another layer into the directory structure, or use symbolic links to group together hosts providing similar service types.

Configuration of pmlogger

The configuration files used by **pmlogger** describe which metrics are to be logged. Groups of metrics may be logged at different intervals to other groups of metrics. Two states, mandatory and advisory, also apply to each group of metrics, defining whether metrics definitely should be logged or not logged, or whether a later advisory definition may change that state.

The mandatory state takes precedence if it is **on** or **off**, causing any subsequent request for a change in advisory state to have no effect. If the mandatory state is **maybe**, then the advisory state determines if logging is enabled or not.

The mandatory states are **on**, **off**, and **maybe**. The advisory states, which only affect metrics that are mandatory **maybe**, are **on** and **off**. Therefore, a metric that is mandatory **maybe** in one definition and advisory **on** in another definition would be logged at the advisory interval. Metrics that are not specified in the **pmlogger** configuration file are mandatory **maybe** and advisory **off** by default and are not logged.

A complete description of the **pmlogger** configuration format can be found on the **pmlogger(1)** man page.

PCP Archive Contents

Once a PCP archive log has been created, the **pmdumplog** utility may be used to display various information about the contents of the archive. For example, start with the following command:

```
pmdumplog -l ${PCP_LOG_DIR}/pmlogger/www.sgi.com/19960731
```

It might produce the following output:

```
Log Label (Log Format Version 1)
Performance metrics from host www.sgi.com
     commencing Wed Jul 31 00:16:34.941 1996
     ending     Thu Aug  1 00:18:01.468 1996
```

The simplest way to discover what performance metrics are contained within a set of archives is to use **pminfo** as shown in *Example 6.1. Using pminfo to Obtain Archive Information*:

Example 6.1. Using pminfo to Obtain Archive Information

```
pminfo -a ${PCP_LOG_DIR}/pmlogger/www.sgi.com/19960731 network.mbuf
network.mbuf.alloc
network.mbuf.typealloc
network.mbuf.clustalloc
network.mbuf.clustfree
network.mbuf.failed
network.mbuf.waited
network.mbuf.drained
```

8.3 Cookbook for Archive Logging

The following sections present a checklist of tasks that may be performed to enable PCP archive logging with minimal effort. For a complete explanation, refer to the other sections in this chapter and the man pages for **pmlogger** and related tools.

8.3.1 Primary Logger

Assume you wish to activate primary archive logging on the PCP collector host **pluto**. Execute the following while logged into **pluto** as the superuser (**root**).

1. Start pmcd and pmlogger:

```
chkconfig pmcd on
chkconfig pmlogger on
${PCP_RC_DIR}/pmcd start
Starting pmcd ...
${PCP_RC_DIR}/pmlogger start
Starting pmlogger ...
```

2. Verify that the primary **pmlogger** instance is running:

```
pcp
Performance Co-Pilot configuration on pluto:

 platform: Linux pluto 3.10.0-0.rc7.64.el7.x86_64 #1 SMP
 hardware: 8 cpus, 2 disks, 23960MB RAM
```

(continues on next page)

(continued from previous page)

```
 timezone: EST-10
     pmcd: Version 4.0.0-1, 8 agents
     pmda: pmcd proc xfs linux mmv infiniband gluster elasticsearch
     pmlogger: primary logger: pluto/20170815.10.00
     pmie: pluto: ${PCP_LOG_DIR}/pmie/pluto/pmie.log
           venus: ${PCP_LOG_DIR}/pmie/venus/pmie.log
```

3. Verify that the archive files are being created in the expected place:

```
ls ${PCP_LOG_DIR}/pmlogger/pluto
20170815.10.00.0
20170815.10.00.index
20170815.10.00.meta
Latest
pmlogger.log
```

4. Verify that no errors are being logged, and the rate of expected growth of the archives:

```
cat ${PCP_LOG_DIR}/pmlogger/pluto/pmlogger.log
Log for pmlogger on pluto started Thu Aug 15 10:00:11 2017

Config parsed
Starting primary logger for host "pluto"
Archive basename: 20170815.00.10

Group [26 metrics] {
            hinv.map.lvname
            ...
            hinv.ncpu
} logged once: 1912 bytes

Group [11 metrics] {
            kernel.all.cpu.user
            ...
            kernel.all.load
} logged every 60 sec: 372 bytes or 0.51 Mbytes/day

...
```

8.3.2 Other Logger Configurations

Assume you wish to create archive logs on the local host for performance metrics collected from the remote host venus. Execute all of the following tasks while logged into the local host as the superuser (**root**).

Procedure 6.1. Creating Archive Logs

1. Create a suitable **pmlogger** configuration file. There are several options:

   - Run the **pmlogconf(1)** utility to generate a configuration file, and (optionally) interactively customize it further to suit local needs.

```
${PCP_BINADM_DIR}/pmlogconf ${PCP_SYSCONF_DIR}/pmlogger/config.venus
Creating config file "${PCP_SYSCONF_DIR}/pmlogger/config.venus" using default␣
↪settings

${PCP_BINADM_DIR}/pmlogconf ${PCP_SYSCONF_DIR}/pmlogger/config.venus
```

(continues on next page)

(continued from previous page)

```
Group: utilization per CPU
Log this group? [n] y
Logging interval? [default]

Group: utilization (usr, sys, idle, ...) over all CPUs
Log this group? [y] y
Logging interval? [default]

Group: per spindle disk activity
Log this group? [n] y

...
```

Do nothing - a default configuration will be created in the following step, using **pmlogconf(1)** probing and automatic file generation based on the metrics available at the remote host. The `${PCP_RC_DIR}/pmlogger` start script handles this.

Manually - create a configuration file with a text editor, or arrange to have one put in place by configuration management tools like Puppet or Chef.

2. Edit `${PCP_PMLOGGERCONTROL_PATH}`, or one of the `${PCP_PMLOGGERCONTROL_PATH}.d` files. Using the line for **remote** as a template, add the following line:

```
venus n n PCP_LOG_DIR/pmlogger/venus -r -T24h10m -c config.venus
```

3. Start **pmlogger**:

```
${PCP_BINADM_DIR}/pmlogger_check
Restarting pmlogger for host "venus" ..... done
```

4. Verify that the **pmlogger** instance is running:

```
pcp
Performance Co-Pilot configuration on pluto:

 platform: Linux pluto 3.10.0-0.rc7.64.el7.x86_64 #1 SMP
 hardware: 8 cpus, 2 disks, 23960MB RAM
 timezone: EST-10
     pmcd: Version 3.8.3-1, 8 agents
     pmda: pmcd proc linux xfs mmv infiniband gluster elasticsearch
     pmlogger: primary logger: pluto/20170815.10.00
           venus.redhat.com: venus/20170815.11.15
pmlc
pmlc> show loggers
The following pmloggers are running on pluto:
        primary (19144) 5141
pmlc> connect 5141
pmlc> status
pmlogger [5141] on host pluto is logging metrics from host venus
log started      Thu Aug 15 11:15:39 2017 (times in local time)
last log entry   Thu Aug 15 11:47:39 2017
current time     Thu Aug 15 11:48:13 2017
log volume       0
log size         146160
```

To create archive logs on the local host for performance metrics collected from multiple remote hosts, repeat the steps in *Procedure 6.1. Creating Archive Logs* for each remote host (each with a new **control** file entry).

8.3.3 Archive Log Administration

Assume the local host has been set up to create archive logs of performance metrics collected from one or more hosts (which may be either the local host or a remote host).

Note: Depending on your platform, the **crontab** entry discussed here may already have been installed for you, as part of the package installation process. In this case, the file **/etc/cron.d/pcp-pmlogger** will exist, and the rest of this section can be skipped.

To activate the maintenance and housekeeping scripts for a collection of archive logs, execute the following tasks while logged into the local host as the superuser (**root**):

1. Augment the **crontab** file for the **pcp** user. For example:

```
crontab -l -u pcp > ${HOME}/crontab.txt
```

2. Edit `${HOME}/crontab.txt`, adding lines similar to those from the sample `${PCP_VAR_DIR}/config/pmlogger/crontab` file for **pmlogger_daily** and **pmlogger_check**; for example:

```
# daily processing of archive logs
10     0     *     *     *    ${PCP_BINADM_DIR}/pmlogger_daily
# every 30 minutes, check pmlogger instances are running
25,55  *     *     *     *    ${PCP_BINADM_DIR}/pmlogger_check
```

3. Make these changes permanent with this command:

```
crontab -u pcp < ${HOME}/crontab.txt
```

8.4 Other Archive Logging Features and Services

Other archive logging features and services include PCP archive folios, manipulating archive logs, primary logger, and using **pmlc**.

8.4.1 PCP Archive Folios

A collection of one or more sets of PCP archive logs may be combined with a control file to produce a PCP archive folio. Archive folios are created using either **mkaf** or the interactive record mode services of various PCP monitor tools (e.g. **pmchart** and **pmrep**).

The automated archive log management services also create an archive folio named **Latest** for each managed **pmlogger** instance, to provide a symbolic name to the most recent archive log. With reference to *Figure 6.1. Archive Log Directory Structure*, this would mean the creation of the folios `${PCP_LOG_DIR}/pmlogger/one/Latest` and `${PCP_LOG_DIR}/pmlogger/two/Latest`.

The **pmafm** utility is completely described in the **pmafm(1)** man page, and provides the interactive commands (single commands may also be executed from the command line) for the following services:

- Checking the integrity of the archives in the folio.
- Displaying information about the component archives.
- Executing PCP tools with their source of performance metrics assigned concurrently to all of the component archives (where the tool supports this), or serially executing the PCP tool once per component archive.

- If the folio was created by a single PCP monitoring tool, replaying all of the archives in the folio with that monitoring tool.
- Restricting the processing to particular archives, or the archives associated with particular hosts.

8.4.2 Manipulating Archive Logs with pmlogextract

The **pmlogextract** tool takes a number of PCP archive logs from a single host and performs the following tasks:

- Merges the archives into a single log, while maintaining the correct time stamps for all values.
- Extracts all metric values within a temporal window that could encompass several archive logs.
- Extracts only a configurable subset of metrics from the archive logs.

See the **pmlogextract(1)** man page for full information on this command.

8.4.3 Summarizing Archive Logs with pmlogsummary

The **pmlogsummary** tool provides statistical summaries of archives, or specific metrics within archives, or specific time windows of interest in a set of archives. These summaries include various averages, minima, maxima, sample counts, histogram bins, and so on.

As an example, for Linux host **pluto**, report on its use of anonymous huge pages - average use, maximum, time at which maximum occurred, total number of samples in the set of archives, and the units used for the values - as shown in *Example 6.2. Using pmlogsummary to Summarize Archive Information*:

Example 6.2. Using pmlogsummary to Summarize Archive Information

```
pmlogsummary -MIly ${PCP_LOG_DIR}/pmlogger/pluto/20170815 mem.util.anonhugepages
Performance metrics from host pluto
  commencing Thu Aug 15 00:10:12.318 2017
  ending     Fri Aug 16 00:10:12.299 2017

mem.util.anonhugepages  7987742.326 8116224.000 15:02:12.300 1437 Kbyte

pminfo -t mem.util.anonhugepages
mem.util.anonhugepages [amount of memory in anonymous huge pages]
```

See the **pmlogsummary(1)** man page for detailed information about this commands many options.

8.4.4 Primary Logger

On each system for which PMCD is active (each PCP collector system), there is an option to have a distinguished instance of the archive logger **pmlogger** (the "primary" logger) launched each time PMCD is started. This may be used to ensure the creation of minimalist archive logs required for ongoing system management and capacity planning in the event of failure of a system where a remote **pmlogger** may be running, or because the preferred archive logger deployment is to activate **pmlogger** on each PCP collector system.

Run the following command as superuser on each PCP collector system where you want to activate the primary **pmlogger**:

```
chkconfig pmlogger on
```

The primary logger launches the next time the `${PCP_RC_DIR}/pmlogger` **start** script runs. If you wish this to happen immediately, follow up with this command:

```
${PCP_BINADM_DIR}/pmlogger_check -V
```

When it is started in this fashion, the `${PCP_PMLOGGERCONTROL_PATH}` file (or one of the `${PCP_PMLOGGERCONTROL_PATH}.d` files) must use the second field of one configuration line to designate the primary logger, and usually will also use the **pmlogger** configuration file `${PCP_SYSCONF_DIR}/pmlogger/config.default` (although the latter is not mandatory).

8.4.5 Using pmlc

You may tailor **pmlogger** dynamically with the **pmlc** command (if it is configured to allow access to this functionality). Normally, the **pmlogger** configuration is read at startup. If you choose to modify the **config** file to change the parameters under which **pmlogger** operates, you must stop and restart the program for your changes to have effect. Alternatively, you may change parameters whenever required by using the **pmlc** interface.

To run the **pmlc** tool, enter:

```
pmlc
```

By default, **pmlc** acts on the primary instance of **pmlogger** on the current host. See the **pmlc(1)** man page for a description of command line options. When it is invoked, **pmlc** presents you with a prompt:

```
pmlc>
```

You may obtain a listing of the available commands by entering a question mark (?) and pressing **Enter**. You see output similar to that in *Example 6.3. Listing Available Commands*:

Example 6.3. Listing Available Commands

```
show loggers [@<host>]           display <pid>s of running pmloggers
connect _logger_id [@<host>]     connect to designated pmlogger
status                           information about connected pmlogger
query metric-list                show logging state of metrics
new volume                       start a new log volume
flush                            flush the log buffers to disk
log { mandatory | advisory } on <interval> _metric-list
log { mandatory | advisory } off _metric-list
log mandatory maybe _metric-list
timezone local|logger|'<timezone>' change reporting timezone
help                               print this help message
quit                               exit from pmlc
_logger_id   is  primary | <pid> | port <n>
_metric-list is  _metric-spec | { _metric-spec ... }
_metric-spec is  <metric-name> | <metric-name> [ <instance> ... ]
```

Here is an example:

```
pmlc
pmlc> show loggers @babylon
The following pmloggers are running on babylon:
       primary (1892)
pmlc> connect 1892 @babylon
pmlc> log advisory on 2 secs disk.dev.read
pmlc> query disk.dev
disk.dev.read
       adv  on  nl       5 min  [131073 or "disk1"]
```

(continues on next page)

(continued from previous page)

```
       adv  on  nl       5 min  [131074 or "disk2"]
pmlc> quit
```

Note: Any changes to the set of logged metrics made via **pmlc** are not saved, and are lost the next time **pmlogger** is started with the same configuration file. Permanent changes are made by modifying the **pmlogger** configuration file(s).

Refer to the **pmlc(1)** and **pmlogger(1)** man pages for complete details.

8.5 Archive Logging Troubleshooting

The following issues concern the creation and use of logs using **pmlogger**.

8.5.1 pmlogger Cannot Write Log

Symptom:

The **pmlogger** utility does not start, and you see this message:

```
__pmLogNewFile: "foo.index" already exists, not over-written
```

Cause:

Archive logs are considered sufficiently precious that **pmlogger** does not empty or overwrite an existing set of archive log files. The log named **foo** actually consists of the physical file **foo.index**, **foo.meta**, and at least one file **foo.N**, where **N** is in the range 0, 1, 2, 3, and so on.

A message similar to the one above is produced when a new **pmlogger** instance encounters one of these files already in existence.

Resolution:

Move the existing archive aside, or if you are sure, remove all of the parts of the archive log. For example, use the following command:

```
rm -f foo.*
```

Then rerun **pmlogger**.

8.5.2 Cannot Find Log

Symptom:

The **pmdumplog** utility, or any tool that can read an archive log, displays this message:

```
Cannot open archive mylog: No such file or directory
```

Cause:

An archive consists of at least three physical files. If the base name for the archive is **mylog**, then the archive actually consists of the physical files **mylog.index**, **mylog.meta**, and at least one file **mylog.N**, where **N** is in the range 0, 1, 2, 3, and so on.

The above message is produced if one or more of the files is missing.

Resolution:

Use this command to check which files the utility is trying to open:

```
ls mylog.*
```

Turn on the internal debug flag **DBG_TRACE_LOG** (**-D** 128) to see which files are being inspected by the **pmOpenLog** routine as shown in the following example:

```
pmdumplog -D 128 -l mylog
```

Locate the missing files and move them all to the same directory, or remove all of the files that are part of the archive, and recreate the archive log.

8.5.3 Primary pmlogger Cannot Start

Symptom:

The primary **pmlogger** cannot be started. A message like the following appears:

```
pmlogger: there is already a primary pmlogger running
```

Cause:

There is either a primary **pmlogger** already running, or the previous primary **pmlogger** was terminated unexpectedly before it could perform its cleanup operations.

Resolution:

If there is already a primary **pmlogger** running and you wish to replace it with a new **pmlogger**, use the **show** command in **pmlc** to determine the process ID of the primary **pmlogger**. The process ID of the primary **pmlogger** appears in parentheses after the word "primary." Send a **SIGINT** signal to the process to shut it down (use either the **kill** command if the platform supports it, or the **pmsignal** command). If the process does not exist, proceed to the manual cleanup described in the paragraph below. If the process did exist, it should now be possible to start the new **pmlogger**.

If **pmlc's show** command displays a process ID for a process that does not exist, a **pmlogger** process was terminated before it could clean up. If it was the primary **pmlogger**, the corresponding control files must be removed before one can start a new primary **pmlogger**. It is a good idea to clean up any spurious control files even if they are not for the primary **pmlogger**.

The control files are kept in `${PCP_TMP_DIR}/pmlogger`. A control file with the process ID of the **pmlogger** as its name is created when the **pmlogger** is started. In addition, the primary **pmlogger** creates a symbolic link named **primary** to its control file.

For the primary **pmlogger**, remove both the symbolic link and the file (corresponding to its process ID) to which the link points. For other **pmloggers**, remove just the process ID file. Do not remove any other files in the directory. If the control file for an active **pmlogger** is removed, **pmlc** is not able to contact it.

8.5.4 Identifying an Active pmlogger Process

Symptom:

You have a PCP archive log that is demonstrably growing, but do not know the identify of the associated **pmlogger** process.

Cause:

The PID is not obvious from the log, or the archive name may not be obvious from the output of the **ps** command.

Resolution:

If the archive basename is **foo**, run the following commands:

```
pmdumplog -l foo
Log Label (Log Format Version 1)
Performance metrics from host gonzo
     commencing Wed Aug  7 00:10:09.214 1996
     ending     Wed Aug  7 16:10:09.155 1996

pminfo -a foo -f pmcd.pmlogger
pmcd.pmlogger.host
     inst [10728 or "10728"] value "gonzo"
pmcd.pmlogger.port
     inst [10728 or "10728"] value 4331
pmcd.pmlogger.archive
     inst [10728 or "10728"] value "/usr/var/adm/pcplog/gonzo/foo"
```

All of the information describing the creator of the archive is revealed and, in particular, the instance identifier for the PMCD metrics (**10728** in the example above) is the PID of the **pmlogger** instance, which may be used to control the process via **pmlc**.

8.5.5 Illegal Label Record

Symptom:

PCP tools report:

```
Illegal label record at start of PCP archive log file.
```

Cause:

The label record at the start of each of the physical archive log files has become either corrupted or one is out of sync with the others.

Resolution:

If you believe the log may have been corrupted, this can be verified using **pmlogcheck**. If corruption is limited to just the label record at the start, the **pmloglabel** can be used to force the labels back in sync with each other, with known-good values that you supply.

Refer to the **pmlogcheck(1)** and **pmloglabel(1)** man pages.

8.5.6 Empty Archive Log Files or pmlogger Exits Immediately

Symptom:

Archive log files are zero size, requested metrics are not being logged, or **pmlogger** exits immediately with no error messages.

Cause:

Either **pmlogger** encountered errors in the configuration file, has not flushed its output buffers yet, or some (or all) metrics specified in the **pmlogger** configuration file have had their state changed to advisory **off** or mandatory **off** via **pmlc**. It is also possible that the logging interval specified in the **pmlogger** configuration file for some or all of the metrics is longer than the period of time you have been waiting since **pmlogger** started.

Resolution:

If **pmlogger** exits immediately with no error messages, check the **pmlogger.log** file in the directory **pmlogger** was started in for any error messages. If **pmlogger** has not yet flushed its buffers, enter one of the following commands (depending on platform support):

```
killall -SIGUSR1 pmlogger
${PCP_BINADM_DIR}/pmsignal -a -s USR1 pmlogger
```

Otherwise, use the **status** command for **pmlc** to interrogate the internal **pmlogger** state of specific metrics.

CHAPTER

NINE

PERFORMANCE CO-PILOT DEPLOYMENT STRATEGIES

Contents

- *Performance Co-Pilot Deployment Strategies*
  - *Basic Deployment*
  - *PCP Collector Deployment*
    - *Principal Server Deployment*
    - *Quality of Service Measurement*
  - *PCP Archive Logger Deployment*
    - *Deployment Options*
    - *Resource Demands for the Deployment Options*
    - *Operational Management*
    - *Exporting PCP Archive Logs*
  - *PCP Inference Engine Deployment*
    - *Deployment Options*
    - *Resource Demands for the Deployment Options*
    - *Operational Management*

Performance Co-Pilot (PCP) is a coordinated suite of tools and utilities allowing you to monitor performance and make automated judgements and initiate actions based on those judgements. PCP is designed to be fully configurable for custom implementation and deployed to meet specific needs in a variety of operational environments.

Because each enterprise and site is different and PCP represents a new way of managing performance information, some discussion of deployment strategies is useful.

The most common use of performance monitoring utilities is a scenario where the PCP tools are executed on a workstation (the PCP monitoring system), while the interesting performance data is collected on remote systems (PCP collector systems) by a number of processes, specifically the Performance Metrics Collection Daemon (PMCD) and the associated Performance Metrics Domain Agents (PMDAs). These processes can execute on both the monitoring system and one or more collector systems, or only on collector systems. However, collector systems are the real objects of performance investigations.

The material in this chapter covers the following areas:

Section 7.1, "*Basic Deployment*", presents the spectrum of deployment architectures at the highest level.

Section 7.2, "*PCP Collector Deployment*", describes alternative deployments for PMCD and the PMDAs.

Section 7.3, "*PCP Archive Logger Deployment*", covers alternative deployments for the **pmlogger** tool.

Section 7.4, "*PCP Inference Engine Deployment*", presents the options that are available for deploying the **pmie** tool.

The options shown in this chapter are merely suggestions. They are not comprehensive, and are intended to demonstrate some possible ways of deploying the PCP tools for specific network topologies and purposes. You are encouraged to use them as the basis for planning your own deployment, consistent with your needs.

9.1 Basic Deployment

In the simplest PCP deployment, one system is configured as both a collector and a monitor, as shown in *Figure 7.1. PCP Deployment for a Single System*. Because some of the PCP monitor tools make extensive use of visualization, this suggests the monitor system should be configured with a graphical display.

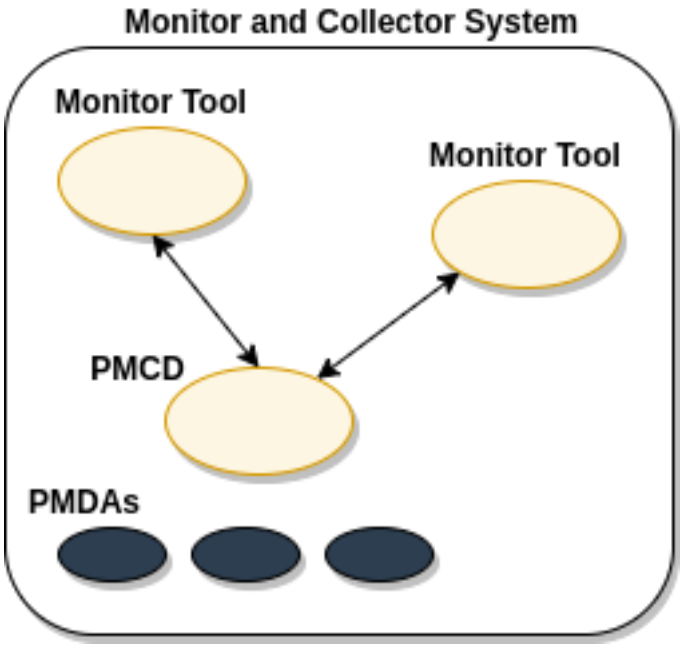

Fig. 1: Figure 7.1. PCP Deployment for a Single System

However, most PCP deployments involve at least two systems. For example, the setup shown in *Figure 7.2. Basic PCP Deployment for Two Systems* would be representative of many common scenarios.

Fig. 2: Figure 7.2. Basic PCP Deployment for Two Systems

But the most common site configuration would include a mixture of systems configured as PCP collectors, as PCP monitors, and as both PCP monitors and collectors, as shown in *Figure 7.3. General PCP Deployment for Multiple Systems*.

With one or more PCP collector systems and one or more PCP monitor systems, there are a number of decisions that need to be made regarding the deployment of PCP services across multiple hosts. For example, in *Figure 7.3. General PCP Deployment for Multiple Systems* there are several ways in which both the inference engine (**pmie**) and the PCP archive logger (**pmlogger**) could be deployed. These options are discussed in the following sections of this chapter.

Fig. 3: Figure 7.3. General PCP Deployment for Multiple Systems

9.2 PCP Collector Deployment

Each PCP collector system must have an active **pmcd** and, typically, a number of PMDAs installed.

9.2.1 Principal Server Deployment

The first hosts selected as PCP collector systems are likely to provide some class of service deemed to be critical to the information processing activities of the enterprise. These hosts include:

- Database servers
- Web servers for an Internet or Intranet presence
- NFS or other central storage server
- A video server
- A supercomputer
- An infrastructure service provider, for example, print, DNS, LDAP, gateway, firewall, router, or mail services
- Any system running a mission-critical application

Your objective may be to improve quality of service on a system functioning as a server for many clients. You wish to identify and repair critical performance bottlenecks and deficiencies in order to maintain maximum performance for clients of the server.

For some of these services, the PCP base product or the PCP add-on packages provide the necessary collector components. Others would require customized PMDA development, as described in the companion *Performance Co-Pilot Programmer's Guide*.

9.2.2 Quality of Service Measurement

Applications and services with a client-server architecture need to monitor performance at both the server side and the client side.

The arrangement in *Figure 7.4. PCP Deployment to Measure Client-Server Quality of Service* illustrates one way of measuring quality of service for client-server applications.

Fig. 4: Figure 7.4. PCP Deployment to Measure Client-Server Quality of Service

The configuration of the PCP collector components on the Application Server System is standard. The new facility is the deployment of some PCP collector components on the Application Client System; this uses a customized PMDA and a generalization of the ICMP "ping" tool as follows:

- The **Client App** is specially developed to periodically make typical requests of the **App Server**, and to measure the response time for these requests (this is an application-specific "ping").
- The PMDA on the Application Client System captures the response time measurements from the **Client App** and exports these into the PCP framework.

At the PCP monitor system, the performance of the system running the **App Server** and the end-user quality of service measurements from the system where the **Client App** is running can be monitored concurrently.

PCP contains a number of examples of this architecture, including the **shping** PMDA for IP-based services (including HTTP), and the **dbping** PMDA for database servers.

The source code for each of these PMDAs is readily available; users and administrators are encouraged to adapt these agents to the needs of the local application environment.

It is possible to exploit this arrangement even further, with these methods:

- Creating new instances of the **Client App** and PMDA to measure service quality for your own mission-critical services.
- Deploying the **Client App** and associated PCP collector components in a number of strategic hosts allows the quality of service over the enterprise's network to be monitored. For example, service can be monitored on the Application Server System, on the same LAN segment as the Application Server System, on the other side of a firewall system, or out in the WAN.

9.3 PCP Archive Logger Deployment

PCP archive logs are created by the **pmlogger** utility, as discussed in Chapter 6, *Archive Logging*. They provide a critical capability to perform retrospective performance analysis, for example, to detect performance regressions, for problem analysis, or to support capacity planning. The following sections discuss the options and trade-offs for **pmlogger** deployment.

9.3.1 Deployment Options

The issue is relatively simple and reduces to "On which host(s) should **pmlogger** be running?" The options are these:

- Run **pmlogger** on each PCP collector system to capture local performance data.
- Run **pmlogger** on some of the PCP monitor systems to capture performance data from remote PCP collector systems.

As an extension of the previous option, designate one system to act as the PCP archive site to run all **pmlogger** instances. This arrangement is shown in *Figure 7.5. Designated PCP Archive Site*.

Fig. 5: Figure 7.5. Designated PCP Archive Site

9.3.2 Resource Demands for the Deployment Options

The **pmlogger** process is very lightweight in terms of computational demand; most of the (very small) CPU cost is associated with extracting performance metrics at the PCP collector system (PMCD and the PMDAs), which are independent of the host on which **pmlogger** is running.

A local **pmlogger** consumes disk bandwidth and disk space on the PCP collector system. A remote **pmlogger** consumes disk space on the site where it is running and network bandwidth between that host and the PCP collector host.

The archive logs typically grow at a rate of anywhere between a few kilobytes (KB) to tens of megabytes (MB) per day, depending on how many performance metrics are logged and the choice of sampling frequencies. There are some advantages in minimizing the number of hosts over which the disk resources for PCP archive logs must be allocated; however, the aggregate requirement is independent of where the **pmlogger** processes are running.

9.3.3 Operational Management

There is an initial administrative cost associated with configuring each **pmlogger** instance, and an ongoing administrative investment to monitor these configurations, perform regular housekeeping (such as rotation, compression, and culling of PCP archive log files), and execute periodic tasks to process the archives (such as nightly performance regression checking with **pmie**).

Many of these tasks are handled by the supplied **pmlogger** administrative tools and scripts, as described in Section 6.2.3, "*Archive Log File Management*". However, the necessity and importance of these tasks favor a centralized pmlogger deployment, as shown in *Figure 7.5. Designated PCP Archive Site*.

9.3.4 Exporting PCP Archive Logs

Collecting PCP archive logs is of little value unless the logs are processed as part of the ongoing performance monitoring and management functions. This processing typically involves the use of the tools on a PCP monitor system, and hence the archive logs may need to be read on a host different from the one they were created on.

NFS mounting is obviously an option, but the PCP tools support random access and both forward and backward temporal motion within an archive log. If an archive is to be subjected to intensive and interactive processing, it may be more efficient to copy the files of the archive log to the PCP monitor system first.

Note: Each PCP archive log consists of at least three separate files (see Section 6.2.3, "*Archive Log File Management*" for details). You must have concurrent access to all of these files before a PCP tool is able to process an archive log correctly.

9.4 PCP Inference Engine Deployment

The **pmie** utility supports automated reasoning about system performance, as discussed in Chapter 5, *Performance Metrics Inference Engine*, and plays a key role in monitoring system performance for both real-time and retrospective analysis, with the performance data being retrieved respectively from a PCP collector system and a PCP archive log.

The following sections discuss the options and trade-offs for **pmie** deployment.

9.4.1 Deployment Options

The issue is relatively simple and reduces to "On which host(s) should **pmie** be running?" You must consider both real-time and retrospective uses, and the options are as follows:

- For real-time analysis, run **pmie** on each PCP collector system to monitor local system performance.
- For real-time analysis, run **pmie** on some of the PCP monitor systems to monitor the performance of remote PCP collector systems.
- For retrospective analysis, run **pmie** on the systems where the PCP archive logs reside. The problem then reduces to **pmlogger** deployment as discussed in Section 7.3, "*PCP Archive Logger Deployment*".
- As an example of the "distributed management with centralized control" philosophy, designate some system to act as the PCP Management Site to run all **pmlogger** and **pmie** instances. This arrangement is shown in *Figure 7.6. PCP Management Site Deployment*.

One **pmie** instance is capable of monitoring multiple PCP collector systems; for example, to evaluate some universal rules that apply to all hosts. At the same time a single PCP collector system may be monitored by multiple **pmie**

instances; for example, for site-specific and universal rule evaluation, or to support both tactical performance management (operations) and strategic performance management (capacity planning). Both situations are depicted in *Figure 7.6. PCP Management Site Deployment*.

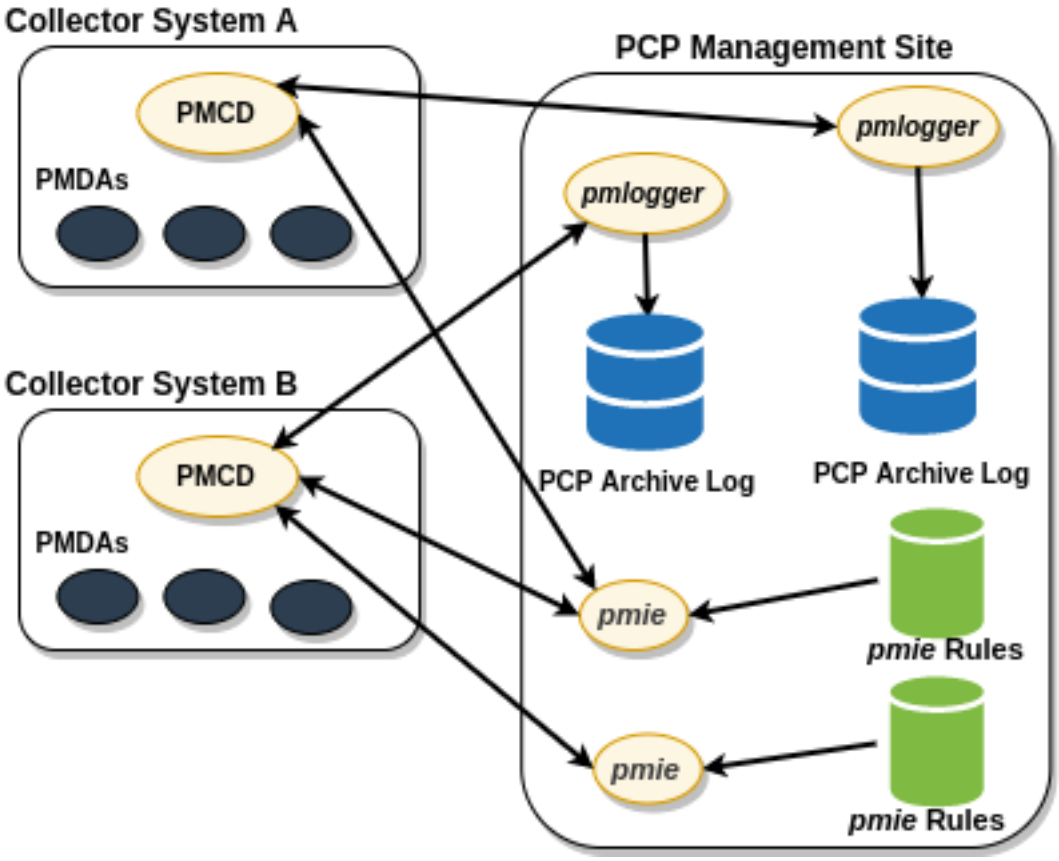

Fig. 6: Figure 7.6. PCP Management Site Deployment

9.4.2 Resource Demands for the Deployment Options

Depending on the complexity of the rule sets, the number of hosts being monitored, and the evaluation frequency, **pmie** may consume CPU cycles significantly above the resources required to simply fetch the values of the performance metrics. If this becomes significant, then real-time deployment of **pmie** away from the PCP collector systems should be considered in order to avoid the "you're part of the problem, not the solution" scenario in terms of CPU utilization on a heavily loaded server.

9.4.3 Operational Management

An initial administrative cost is associated with configuring each **pmie** instance, particularly in the development of the rule sets that accurately capture and classify "good" versus "bad" performance in your environment. These rule sets almost always involve some site-specific knowledge, particularly in respect to the "normal" levels of activity and resource consumption. The pmieconf tool (see Section 5.7, "*Creating pmie Rules with pmieconf*") may be used to help develop localized rules based upon parameterized templates covering many common performance scenarios. In complex environments, customizing these rules may occur over an extended period and require considerable performance analysis insight.

One of the functions of **pmie** provides for continual detection of adverse performance and the automatic generation of alarms (visible, audible, e-mail, pager, and so on). Uncontrolled deployment of this alarm initiating capability throughout the enterprise may cause havoc.

These considerations favor a centralized **pmie** deployment at a small number of PCP monitor sites, or in a PCP Management Site as shown in *Figure 7.6. PCP Management Site Deployment*.

However, it is most likely that knowledgeable users with specific needs may find a local deployment of **pmie** most useful to track some particular class of service difficulty or resource utilization. In these cases, the alarm propagation is unlikely to be required or is confined to the system on which **pmie** is running.

Configuration and management of a number of **pmie** instances is made much easier with the scripts and control files described in Section 5.8, "*Management of pmie Processes*".

CHAPTER

TEN

CUSTOMIZING AND EXTENDING PCP SERVICES

Contents

- *Customizing and Extending PCP Services*
  - *PMDA Customization*
    - *Customizing the Summary PMDA*
  - *PCP Tool Customization*
    - *Archive Logging Customization*
    - *Inference Engine Customization*
  - *PMNS Management*
    - *PMNS Processing Framework*
    - *PMNS Syntax*
  - *PMDA Development*
  - *PCP Tool Development*

Performance Co-Pilot (PCP) has been developed to be fully extensible. The following sections summarize the various facilities provided to allow you to extend and customize PCP for your site:

Section 8.1, "*PMDA Customization*", describes the procedure for customizing the summary PMDA to export derived metrics formed by aggregation of base PCP metrics from one or more collector hosts.

Section 8.2, "*PCP Tool Customization*", describes the various options available for customizing and extending the basic PCP tools.

Section 8.3, "*PMNS Management*", covers the concepts and tools provided for updating the PMNS (Performance Metrics Name Space).

Section 8.4, "*PMDA Development*", details where to find further information to assist in the development of new PMDAs to extend the range of performance metrics available through the PCP infrastructure.

Section 8.5, "*PCP Tool Development*", outlines how new tools may be developed to process performance data from the PCP infrastructure.

10.1 PMDA Customization

The generic procedures for installing and activating the optional PMDAs have been described in Section 2.3, "*Managing Optional PMDAs*". In some cases, these procedures prompt the user for information based upon the local system or network configuration, application deployment, or processing profile to customize the PMDA and hence the performance metrics it exports.

The summary PMDA is a special case that warrants further discussion.

10.1.1 Customizing the Summary PMDA

The summary PMDA exports performance metrics derived from performance metrics made available by other PMDAs. It is described completely in the **pmdasummary(1)** man page.

The summary PMDA consists of two processes:

1. **pmie** process

   Periodically samples the base metrics and compute values for the derived metrics. This dedicated instance of the PCP **pmie** inference engine is launched with special command line arguments by the main process. See Section 5.1, "*Introduction to pmie*", for a complete discussion of the **pmie** feature set.

2. **main** process

   Reads and buffers the values computed by the **pmie** process and makes them available to the Performance Metrics Collection Daemon (PMCD).

All of the metrics exported by the summary PMDA have a singular instance and the values are instantaneous; the exported value is the correct value as of the last time the corresponding expression was evaluated by the **pmie** process.

The summary PMDA resides in the `${PCP_PMDAS_DIR}/summary` directory and may be installed with a default configuration by following the steps described in Section 2.3.1, "*PMDA Installation on a PCP Collector Host*".

Alternatively, you may customize the summary PMDA to export your own derived performance metrics by following the steps in *Procedure 8.1. Customizing the Summary PMDA*:

Procedure 8.1. Customizing the Summary PMDA

1. Check that the symbolic constant **SYSSUMMARY** is defined in the `${PCP_VAR_DIR}/pmns/stdpmid` file. If it is not, perform the postinstall update of this file, as superuser:

   ```
   cd ${PCP_VAR_DIR}/pmns ./Make.stdpmid
   ```

2. Choose Performance Metric Name Space (PMNS) names for the new metrics. These must begin with **summary** and follow the rules described in the **pmns(5)** man page. For example, you might use **summary.fs.cache_write** and **summary.fs.cache_hit**.

3. Edit the **pmns** file in the `${PCP_PMDAS_DIR}/summary` directory to add the new metric names in the format described in the **pmns(5)** man page. You must choose a unique performance metric identifier (PMID) for each metric. In the **pmns** file, these appear as **SYSSUMMARY:0:x**. The value of x is arbitrary in the range 0 to 1023 and unique in this file. Refer to Section 8.3, "*PMNS Management*", for a further explanation of the rules governing PMNS updates.

   For example:

   ```
   summary {
             cpu
             disk
             netif
   ```

(continues on next page)

(continued from previous page)

```
          fs              /*new*/
}
summary.fs {
          cache_write     SYSSUMMARY:0:10
          cache_hit       SYSSUMMARY:0:11
}
```

4. Use the local test PMNS **root** and validate that the PMNS changes are correct.

   For example, enter this command:

```
pminfo -n root -m summary.fs
```

   You see output similar to the following:

```
summary.fs.cache_write PMID: 27.0.10
summary.fs.cache_hit PMID: 27.0.11
```

5. Edit the `${PCP_PMDAS_DIR}/summary/expr.pmie` file to add new **pmie** expressions. If the name to the left of the assignment operator (=) is one of the PMNS names, then the **pmie** expression to the right will be evaluated and returned by the summary PMDA. The expression must return a numeric value. Additional description of the **pmie** expression syntax may be found in Section 5.3, "*Specification Language for pmie*".

   For example, consider this expression:

```
// filesystem buffer cache hit percentages
prefix = "kernel.all.io";          // macro, not exported
summary.fs.cache_write =
                100 - 100 * $prefix.bwrite / $prefix.lwrite;
summary.fs.cache_hit =
                100 - 100 * $prefix.bread / $prefix.lread;
```

6. Run **pmie** in debug mode to verify that the expressions are being evaluated correctly, and the values make sense.

   For example, enter this command:

```
pmie -t 2sec -v expr.pmie
```

   You see output similar to the following:

```
summary.fs.cache_write:      ?
summary.fs.cache_hit:      ?
summary.fs.cache_write:  45.83
summary.fs.cache_hit:   83.2
summary.fs.cache_write:  39.22
summary.fs.cache_hit:  84.51
```

7. Install the new PMDA.

   From the `${PCP_PMDAS_DIR}/summary` directory, use this command:

```
./Install
```

   You see the following output:

```
Interval between summary expression evaluation (seconds)? [10] 10
Updating the Performance Metrics Name Space...
```

(continues on next page)

(continued from previous page)

```
Installing pmchart view(s) ...
Terminate PMDA if already installed ...
Installing files ..
Updating the PMCD control file, and notifying PMCD ...
Wait 15 seconds for the agent to initialize ...
Check summary metrics have appeared ... 8 metrics and 8 values
```

8. Check the metrics.

   For example, enter this command:

```
pmval -t 5sec -s 8 summary.fs.cache_write
```

   You see a response similar to the following:

```
metric:    summary.fs.cache_write
host:      localhost
semantics: instantaneous value
units:     none
samples:   8
interval:  5.00 sec
63.60132158590308
62.71878646441073
62.71878646441073
58.73968492123031
58.73968492123031
65.33822758259046
65.33822758259046
72.6099706744868
```

   Note that the values are being sampled here by **pmval** every 5 seconds, but **pmie** is passing only new values to the summary PMDA every 10 seconds. Both rates could be changed to suit the dynamics of your new metrics.

9. You may now create **pmchart** views, **pmie** rules, and **pmlogger** configurations to monitor and archive your new performance metrics.

10.2 PCP Tool Customization

Performance Co-Pilot (PCP) has been designed and implemented with a philosophy that embraces the notion of toolkits and encourages extensibility.

In most cases, the PCP tools provide orthogonal services, based on external configuration files. It is the creation of new and modified configuration files that enables PCP users to customize tools quickly and meet the needs of the local environment, in many cases allowing personal preferences to be established for individual users on the same PCP monitor system.

The material in this section is intended to act as a checklist of pointers to detailed documentation found elsewhere in this guide, in the man pages, and in the files that are made available as part of the PCP installation.

10.2.1 Archive Logging Customization

The PCP archive logger is presented in Chapter 6, *Archive Logging*, and documented in the **pmlogger(1)** man page.

The following global files and directories influence the behavior of **pmlogger**:

`${PCP_SYSCONF_DIR}/pmlogger`

Enable/disable state for the primary logger facility using this command:

```
chkconfig pmlogger on
```

`${PCP_SYSCONF_DIR}/pmlogger/config.default`

The default **pmlogger** configuration file that is used for the primary logger when this facility is enabled.

`${PCP_VAR_DIR}/config/pmlogconf/tools`

Every PCP tool with a fixed group of performance metrics contributes a **pmlogconf** configuration file that includes each of the performance metrics used in the tool, for example, `${PCP_VAR_DIR}/config/pmlogconf/pmstat` for **pmstat**.

`${PCP_PMLOGGERCONTROL_PATH}` or `${PCP_PMLOGGERCONTROL_PATH}.d` files

Defines which PCP collector hosts require **pmlogger** to be launched on the local host, where the configuration file comes from, where the archive log files should be created, and **pmlogger** startup options.

These **control** files support the starting and stopping of multiple **pmlogger** instances that monitor local or remote hosts.

`/etc/cron.d/pcp-pmlogger` or `${PCP_VAR_DIR}/config/pmlogger/crontab`

Default **crontab** entries that may be merged with the **crontab** entries for the **pcp** user to schedule the periodic execution of the archive log management scripts, for example, **pmlogger_daily**.

`${PCP_LOG_DIR}/pmlogger/somehost`

The default behavior of the archive log management scripts create archive log files for the host *somehost* in this directory.

`${PCP_LOG_DIR}/pmlogger/somehost/Latest`

A PCP archive folio for the most recent archive for the host *somehost*. This folio is created and maintained by the **cron**-driven periodic archive log management scripts, for example, **pmlogger_check**. Archive folios may be processed with the **pmafm** tool.

10.2.2 Inference Engine Customization

The PCP inference engine is presented in Chapter 5, *Performance Metrics Inference Engine*, and documented in the **pmie(1)** man page.

The following global files and directories influence the behavior of **pmie**:

`${PCP_SYSCONF_DIR}/pmie`

Controls the pmie daemon facility. Enable using this command:

```
chkconfig pmie on
```

`${PCP_SYSCONF_DIR}/pmie/config.default`

The **pmie** configuration file that is used for monitoring the local host when the **pmie** daemon facility is enabled in the default configuration. This file is created using **pmieconf** the first time the daemon facility is activated.

`${PCP_PMIECONTROL_PATH}` and `${PCP_PMIECONTROL_PATH}.d` files

Defines which PCP collector hosts require a daemon **pmie** to be monitoring from the local host, where the configuration files comes from, where the **pmie** log file should be created, and **pmie** startup options.

These **control** files support the starting and stopping of multiple **pmie** instances that are each monitoring one or more hosts.

`${PCP_VAR_DIR}/config/pmieconf/*/*`

Each **pmieconf** rule definition can be found below one of these subdirectories.

`/etc/cron.d/pcp-pmie` or `${PCP_VAR_DIR}/config/pmie/crontab`

Default **crontab** entries that may be merged with the **crontab** entries for the **pcp** user to schedule the periodic execution of the **pmie_check** and **pmie_daily** scripts, for verifying that **pmie** instances are running and logs rotated.

`${PCP_LOG_DIR}/pmie/somehost`

The default behavior of the `${PCP_RC_DIR}/pmie` startup scripts create **pmie** log files for the host *somehost* in this directory.

`pmie_check` and `pmie_daily`

These commands are similar to the **pmlogger** support scripts, **pmlogger_check** and **pmlogger_daily**.

`${PCP_TMP_DIR}/pmie`

The statistics that **pmie** gathers are maintained in binary data structure files. These files can be found in the `${PCP_TMP_DIR}/pmie` directory.

pmcd.pmie metrics

The PMCD PMDA exports information about executing **pmie** processes and their progress in terms of rule evaluations and action execution rates.

If **pmie** is running on a system with a PCP collector deployment, the **pmcd** PMDA exports these metrics via the **pmcd.pmie** group of metrics.

10.3 PMNS Management

This section describes the syntax, semantics, and processing framework for the external specification of a Performance Metrics Name Space (PMNS) as it might be loaded by the PMAPI routine **pmLoadNameSpace**; see the **pmLoadNameSpace(3)** man page. This is usually done only by **pmcd**, except in rare circumstances such as Section 8.1.1, "*Customizing the Summary PMDA*".

The PMNS specification is a simple text source file that can be edited easily. For reasons of efficiency, a binary format is also supported; the utility **pmnscomp** translates the ASCII source format into binary format; see the **pmnscomp(1)** man page.

10.3.1 PMNS Processing Framework

The PMNS specification is initially passed through **pmcpp(1)**. This means the following facilities may be used in the specification:

- C-style comments
- **#include** directives
- **#define** directives and macro substitution
- Conditional processing with **#ifdef, #ifndef, #endif**, and **#undef**

When **pmcpp(1)** is executed, the standard include directories are the current directory and `${PCP_VAR_DIR}/pmns`, where some standard macros and default specifications may be found.

10.3.2 PMNS Syntax

Every PMNS is tree structured. The paths to the leaf nodes are the performance metric names. The general syntax for a non-leaf node in PMNS is as follows:

```
pathname {
        name    [pmid]
        ...
}
```

Here **pathname** is the full pathname from the root of the PMNS to this non-leaf node, with each component in the path separated by a period. The root node for the PMNS has the special name **root**, but the prefix string **root**. must be omitted from all other **pathnames**.

For example, refer to the PMNS shown in *Figure 8.1. Small Performance Metrics Name Space (PMNS)*. The correct pathname for the rightmost non-leaf node is **cpu.utilization**, not **root.cpu.utilization**.

Fig. 1: Figure 8.1. Small Performance Metrics Name Space (PMNS)

Each component in the pathname must begin with an alphabetic character and be followed by zero or more alphanumeric characters or the underscore (_) character. For alphabetic characters in a component, uppercase and lowercase are significant.

Non-leaf nodes in the PMNS may be defined in any order desired. The descendent nodes are defined by the set of **names**, relative to the pathname of their parent non-leaf node. For descendent nodes, leaf nodes have a **pmid** specification, but non-leaf nodes do not.

The syntax for the **pmid** specification was chosen to help manage the allocation of Performance Metric IDs (PMIDs) across disjoint and autonomous domains of administration and implementation. Each **pmid** consists of three integers separated by colons, for example, **14:27:11**. This is intended to mirror the implementation hierarchy of performance metrics. The first integer identifies the domain in which the performance metric lies. Within a domain, related metrics are often grouped into clusters. The second integer identifies the cluster, and the third integer, the metric within the cluster.

The PMNS specification for *Figure 8.1. Small Performance Metrics Name Space (PMNS)* is shown in *Example 8.1. PMNS Specification*:

Example 8.1. PMNS Specification

```
/*
* PMNS Specification
*/
#define KERNEL 1
root {
       network
       cpu
}
#define NETWORK 26
network {
       interrupts    KERNEL:NETWORK:1
       packets
}
network.packets {
       in     KERNEL:NETWORK:35
       out    KERNEL:NETWORK:36
}
#define CPU 10
cpu {
       syscalls    KERNEL:CPU:10
       utilization
}
#define USER 20
#define SYSTEM 21
#define IDLE 22
cpu.utilization {
       user    KERNEL:CPU:USER
       sys     KERNEL:CPU:SYSTEM
       idle    KERNEL:CPU:IDLE
}
```

For complete documentation of the PMNS and associated utilities, see the **pmns(5), pmnsadd(1), pmnsdel(1)** and **pmnsmerge(1)** man pages.

10.4 PMDA Development

Performance Co-Pilot (PCP) is designed to be extensible at the collector site.

Application developers are encouraged to create new PMDAs to export performance metrics from the applications and service layers that are particularly relevant to a specific site, application suite, or processing environment.

These PMDAs use the routines of the **libpcp_pmda** library, which is discussed in detail in the *Performance Co-Pilot Programmer's Guide*.

10.5 PCP Tool Development

Performance Co-Pilot (PCP) is designed to be extensible at the monitor site.

Application developers are encouraged to create new PCP client applications to monitor or display performance metrics in a manner that is particularly relevant to a specific site, application suite, or processing environment.

Client applications use the routines of the PMAPI (performance metrics application programming interface) described in the *Performance Co-Pilot Programmer's Guide*. At the time of writing, native PMAPI interfaces are available for the C, C++ and Python languages.

CHAPTER

ELEVEN

FAST, SCALABLE TIME SERIES QUERYING - PMSERIES

Contents

- *Fast, Scalable Time Series Querying - pmseries*
  - *Introduction to pmseries*
  - *Timeseries Queries*
  - *Metadata Qualifiers and Metadata Operators*
    - *Boolean operators*
    - *String operators*
    - *Relational operators (numeric label values only)*
    - *Logical operators*
  - *Time Specification*
    - *Sample count*
    - *Sample interval*
    - *Time window*
    - *Time zones*
  - *Expressions*
    - *Arithmetic Operators*
    - *Functions*
    - *Function Reference*
    - *Compatibility*
  - *Timeseries Options*
    - *Timeseries Metadata*
    - *Timeseries Sources*
    - *Timeseries Loading*
    - *Options*
    - *Examples*
  - *PCP Environment*

– *PCP Grafana Plugin*

pmseries is a fast, scalable time series querying which displays information about performance metrics.

The major sections in this chapter are as follows:

Section 9.1, "*Introduction to pmseries*", provides an introduction to the concepts and working of **pmseries**.

Section 9.2, "*Timeseries Queries*", explains how query expressions are formed using the **pmseries** query language.

Section 9.3, "*Metadata Qualifiers and Metadata Operators*", explains various metadata properties.

Section 9.4, "*Time Specification*", specifies a specific time window of interest.

Section 9.5, "*Expressions*", explains the various arithmetic operators, functions, function references as well as their compatibility supported by **pmseries**.

Section 9.6, "*Timeseries Options*", explains the various timeseries options requested to **pmseries** using command line.

Section 9.7, "*PCP Environment*", describes environment variables used to parameterize the file and directory names used by PCP.

Section 9.8, "*PCP Grafana Plugin*", explains the PCP Redis data source and lays out the path to the PCP Grafana Plugin.

11.1 Introduction to pmseries

pmseries displays various types of information about performance metrics available through the scalable timeseries facilities of the Performance Co-Pilot (PCP) using the Redis distributed data store.

By default **pmseries** communicates with a local redis-server(1), however the **-h** and **-p** options can be used to specify an alternate Redis instance. If this instance is a node of a Redis cluster, all other instances in the cluster will be discovered and used automatically.

pmseries runs in several different modes - either querying timeseries identifiers, metadata or values (already stored in Redis), or manually loading timeseries into Redis. The latter mode is seldom used, however, since pmproxy(1) will automatically perform this function for local pmlogger(1) instances, when running in its default time series mode.

Without command line options specifying otherwise, **pmseries** will issue a timeseries query to find matching timeseries and values. All timeseries are identified using a unique SHA-1 hash which is always displayed in a 40-hexdigit human readable form. These hashes are formed using the metadata associated with every metric.

Importantly, this includes all metric metadata (labels, names, descriptors). Metric labels in particular are (as far as possible) unique for every machine - on Linux for example the labels associated with every metric include the unique `/etc/machine-id` , the hostname, domainname, and other automatically generated machine labels, as well as any administrator-defined labels from `/etc/pcp/labels` . These labels can be reported with pminfo(1) and the *pmcd.labels* metric.

See pmLookupLabels(3), pmLookupInDom(3), pmLookupName(3) and pmLookupDesc(3) for detailed information about metric labels and other metric metadata used in each timeseries identifier hash calculation.

The timeseries identifiers provide a higher level (and machine independent) identifier than the traditional PCP performance metric identifiers (pmID), instance domain identifiers (pmInDom) and metric names. See PCPIntro(1) for more details about these traditional identifiers. However, **pmseries** uses timeseries identifiers in much the same way that pminfo(1) uses the lower level indom, metric identifiers and metric names.

The default mode of **pmseries** operation (i.e. with no command line options) depends on the arguments it is presented. If all non-option arguments appear to be timeseries identifiers (in 40 hex digit form) **pmseries** will report metadata for these timeseries - refer to the **-a** option for details. Otherwise, the parameters will be treated as a timeseries query.

11.2 Timeseries Queries

Query expressions are formed using the **pmseries** query language described below, but can be as simple as a metric name.

The following is an example of querying timeseries from all hosts that match a metric name pattern (globbed):

```
$ pmseries kernel.all.cpu*
1d7b0bb3f6aec0f49c54f5210885464a53629b60
379db729afd63fb9eff436625bd6c55a7adc5cfd
3dd3b45bb05f96636043e5d58b52b441ce542285
[...]
ed2bf325ff6dc7589ec966698e5404b67252306a
dcb2a032a308b5717bf605ba8f8737e9c6e1ed19
```

To identify timeseries expression operands, the query language uses the general syntax:

```
[metric.name] '{metadata qualifiers}' '[time specification]'
```

The *metric.name* component restricts the timeseries query to any matching PCP metric name (the list of metric names for a PCP archive or live host is reported by pminfo(1) with no arguments beyond – **host** or – **archive**). The **pmseries** syntax extends on that of **pminfo** and allows for glob(7) based pattern matching within the metric name. The above describes operands available as the leaves of **pmseries** expressions, which may include functions, arithmetic operators and other features. See the *EXPRESSIONS* section below for further details.

11.3 Metadata Qualifiers and Metadata Operators

Metadata qualifiers are enclosed by "curly" braces (**{}**), and further restrict the query results to timeseries operands with various metadata properties. These qualifiers are based on metric or instance names, and metric label values, and take the general form *metadata.name* OPERATOR *value* , such as:

```
instance.name == "cpu0"
metric.name != "kernel.all.pswitch"
```

When using label names, the metadata qualifier is optional and can be dropped, such as:

```
label.hostname == "www.acme.com"
hostname == "www.acme.com"
```

For metric and instance names only the string operators apply, but for metric label values all operators are available. The set of available operators is:

11.3.1 Boolean operators

All string (label, metrics and instances) and numeric (label) values can be tested for equality ("==") or inequality ("!=").

11.3.2 String operators

Strings can be subject to pattern matching in the form of glob matching (“~~”), regular expression matching (“=~”), and regular expression non-matching (“!~”). The “:” operator is equivalent to “~~” - i.e., regular expression matching.

11.3.3 Relational operators (numeric label values only)

Numeric label values can be subject to the less than (“<”), greater than (“>”), less than or equal (“<=”), greater than or equal (“>=”), equal (“==”) and not equal (“!=”) operators.

11.3.4 Logical operators

Multiple metadata qualifiers can be combined with the logical operators for AND (“&&”) and OR (“||”) as in many programming languages. The comma (“,”) character is equivalent to logical AND (“&&”).

11.4 Time Specification

The final (optional) component of a query allows the user to specify a specific time window of interest. Any time specification will result in values being returned for all matching timeseries only for the time window specified.

The specification is “square” bracket (**[]**) enclosed, and consists of one or more comma-separated components. Each component specifies some aspect related to time, taking the general form: **keyword** : *value* , such as:

```
samples:10
```

11.4.1 Sample count

The number of samples to return, specified via either the **samples** or (equivalent) **count** keyword. The *value* provided must be a positive integer. If no end time is explicitly set (see “Time window” later) then the most recent samples will be returned.

11.4.2 Sample interval

An interval between successive samples can be requested using the **interval** or (equivalent) **delta** keyword. The *value* provided should be either a numeric or string value that will be parsed by pmParseInterval(3), such as **5** (seconds) or **2min** (minutes).

11.4.3 Time window

Start and end times, and alignments, affecting the returned values. The keywords match the parameters to the pmParseTimeWindow(3) function which will be used to parse them, and are: **start** or (equivalent) **begin** , **finish** or (equivalent) **end** , **align** and **offset**.

11.4.4 Time zones

The resulting timestamps can be returned having been evaluated for a specific timezone, using the **timezone** or **hostzone** keywords. The *value* associated with **timezone** will be interpreted by pmNewZone(3). A **true** or **false** value should be associated with **hostzone** , and when set to **true** this has the same effect as described by pmNewContextZone(3).

11.5 Expressions

As described above, operands are the leaves of a query expression tree.

```
[metric.name] '{metadata qualifiers}' '[time specification]'
```

Note in most of the query expression examples below, the *metadata qualifiers* have been omitted for brevity. In all cases, multiple time series may qualify, particularly for the **hostname** label.

In the simple case, a query expression consists of a single operand and may just be a metric name. In the more general case, a query expression is either an operand or the argument to a function, or two operands in a binary arithmetic or logical expression. Most functions take a single argument (an expression), though some require additional arguments, e.g. **rescale**.

```
operand | expr operator expr | func(expr[, arg])
```

This grammar shows expressions may be nested, e.g. using the addition (**+**) operator as an example,

```
func1(func2(expr))
func1(expr) + func2(expr)
expr + func(expr)
func(expr) + expr
expr + expr
```

Rules governing compatibility of operands in an expression generally depend on the function and/or operators and are described below individually. An important rule is that if any time windows are specified, then all operands must cover the same number of samples, though the time windows may differ individually. If no time windows or sample counts are given, then **pmseries** will return a series identifier (SID) instead of a series of timestamps and values. This SID may be used in subsequent `/series/values?series= SID` REST API calls, along with a specific time window.

11.5.1 Arithmetic Operators

pmseries support addition, subtraction, division and multiplication on each value in the time series of a binary pair of operands. No unary or ternary operators are supported (yet). In all cases, the instance domain and the number of samples of time series operands must be the same. The metadata (units and dimensions) must also be compatible. Depending on the function, the result will usually have the same instance domain and (unless noted otherwise), the same units as the operands. The metadata dimensions (space, time, count) of the result may differ (see below).

Expression operands may have different qualifiers, e.g. you can perform binary arithmetic on metrics qualified by different labels (such as **hostname**), or metric names. For example, to add the two most recent samples of the process context switch (pswitch) counter metric for hosts **node88** and **node89**, and then perform rate conversion:

```
$ pmseries 'rate(kernel.all.pswitch{hostname:node88}[count:2] +
                 kernel.all.pswitch{hostname:node89}[count:2])'
```

(continues on next page)

(continued from previous page)

```
1cf1a85d5978640ef94c68264d3ae8866cc11f7c
   [Tue Nov 10 14:39:48.771868000 2020] 71.257509␣
↪8e0a59304eb99237b89593a3e839b5bb8b9a9924
```

Note the resulting time series of values has one less sample than the expression operand passed to the **rate** function.

Other rules for arithmetic expressions:

1. If both operands have the semantics of a counter, then only addition and subtraction are allowed.
2. If the left operand is a counter and the right operand is not, then only multiplication or division are allowed
3. If the left operand is not a counter and the right operand is a counter, then only multiplication is allowed.
4. Addition and subtraction - the dimensions of the result are the same as the dimensions of the operands.
5. Multiplication - the dimensions of the result are the sum of the dimensions of the operands.
6. Division - the dimensions of the result are the difference of the dimensions of the operands.

11.5.2 Functions

Expression functions operate on vectors of time series values, and may be nested with other functions or expressions as described above. When an operand has multiple instances, there will generally be one result for each series of instances. For example, the result for

```
$ pmseries 'min(kernel.all.load[count:100])'
```

will be the smallest value of the 100 most recent samples, treating each of the three load average instances as a separate time series. As an example, for the two most recent samples for each of the three instances of the load average metric:

```
$ pmseries 'kernel.all.load[count:2]'
726a325c4c1ba4339ecffcdebd240f441ea77848
    [Tue Nov 10 11:52:30.833379000 2020] 1.100000e+00␣
↪a7c96e5e2e0431a12279756d11590fa9fed8f306
    [Tue Nov 10 11:52:30.833379000 2020] 9.900000e-01␣
↪ee9b506935fd0976a893dc27242926f49326b9a1
    [Tue Nov 10 11:52:30.833379000 2020] 1.070000e+00␣
↪d5e1c360d13064c461169091997e1e8be7488133
    [Tue Nov 10 11:52:20.827134000 2020] 1.120000e+00␣
↪a7c96e5e2e0431a12279756d11590fa9fed8f306
    [Tue Nov 10 11:52:20.827134000 2020] 9.900000e-01␣
↪ee9b506935fd0976a893dc27242926f49326b9a1
    [Tue Nov 10 11:52:20.827134000 2020] 1.070000e+00␣
↪d5e1c360d13064c461169091997e1e8be7488133
```

Using the **min** function :

```
$ pmseries 'min(kernel.all.load[count:2])'
11b965bc5f9598034ed9139fb3a78c6c0b7065ba
    [Tue Nov 10 11:52:30.833379000 2020] 1.100000e+00␣
↪a7c96e5e2e0431a12279756d11590fa9fed8f306
    [Tue Nov 10 11:52:30.833379000 2020] 9.900000e-01␣
↪ee9b506935fd0976a893dc27242926f49326b9a1
    [Tue Nov 10 11:52:30.833379000 2020] 1.070000e+00␣
↪d5e1c360d13064c461169091997e1e8be7488133
```

For singular metrics (with no instance domain), a single value will result, e.g. for the five most recent samples of the context switching metric:

```
$ pmseries 'kernel.all.pswitch[count:5]'
d7832c4fba33bcc980b1a1b614e0508043288480
    [Tue Nov 10 12:44:59.380666000 2020] 460774294
    [Tue Nov 10 12:44:49.382070000 2020] 460747232
    [Tue Nov 10 12:44:39.378545000 2020] 460722370
    [Tue Nov 10 12:44:29.379029000 2020] 460697388
    [Tue Nov 10 12:44:19.379096000 2020] 460657412

$ pmseries 'min(kernel.all.pswitch[count:5])'
1b6e92fb5bc012372f54452734dd03f0f131fa06
    [Tue Nov 10 12:44:19.379096000 2020] 460657412␣
↪d7832c4fba33bcc980b1a1b614e0508043288480
```

Future versions of **pmseries** may provide functions that perform aggregation, interpolation, filtering or transforms in other ways, e.g. across instances instead of time.

11.5.3 Function Reference

- **max** (*expr*) : The maximum value in the time series for each instance of *expr*.
- **min** (*expr*) : The minimum value in the time series for each instance of *expr*.
- **rate** (*expr*) : The rate with respect to time of each sample. The given *expr* must have counter semantics and the result will have **instant** semantics (the time dimension reduced by one). In addition, the result will have one less sample than the operand - this is because the first sample cannot be rate converted (two samples are required).
- **rescale** (*expr* , *scale*) rescale the values in the time series for each instance of *expr* to scale (units). Note that *expr* should have **instant** or **discrete** semantics (not **counter** - rate conversion should be done first if needed). The time, space and count dimensions between *expr* and *scale* must be compatible. Example: rate convert the read throughput counter for each disk instance and then rescale to mbytes per second. Note the native units of **disk.dev.read_bytes** is a **counter** of kbytes read from each device instance since boot.

  ```
  $ pmseries 'rescale(rate(disk.dev.read_bytes[count:4]), "mbytes/s")'
  ```

- **abs** (*expr*) : The absolute value of each value in the time series for each instance of *expr* . This has no effect if the type of *expr* is unsigned.
- **floor** (*expr*) : Rounded down to the nearest integer value of the time series for each instance of *expr*.
- **round** (*expr*) : Rounded up or down to the nearest integer for each value in the time series for each instance of *expr*.
- **log** (*expr*) : Logarithm of the values in the time series for each instance of *expr*.
- **sqrt** (*expr*) : Square root of the values in the time series for each instance of *expr*.

11.5.4 Compatibility

All operands in an expression must have the same number of samples, but not necessarily the same time window. e.g. you could subtract some metric time series from today from that of yesterday by giving different time windows and different metrics or qualifiers, ensuring the same number of samples are given as the operands.

Operands in an expression must either all have a time window, or none. If no operands have a time window, then instead of a series of time stamps and values, the result will be a time series identifier (*SID*) that may be passed to the `/series/values?series= SID` REST API function, along with a time window. For further details, see PMWEBAPI(3).

If the semantics of both operands in an arithmetic expression are not counter (i.e. **PM_SEM_INSTANT** or **PM_SEM_DISCRETE**) then the result will have semantics **PM_SEM_INSTANT** unless both operands are **PM_SEM_DISCRETE** in which case the result is also **PM_SEM_DISCRETE**.

11.6 Timeseries Options

11.6.1 Timeseries Metadata

Using command line options, **pmseries** can be requested to provide metadata (metric names, instance names, labels, descriptors) associated with either individual timeseries or a group of timeseries, for example:

```
$ pmseries -a dcb2a032a308b5717bf605ba8f8737e9c6e1ed19

dcb2a032a308b5717bf605ba8f8737e9c6e1ed19
    PMID: 60.0.21
    Data Type: 64-bit unsigned int  InDom: PM_INDOM_NULL 0xffffffff
    Semantics: counter  Units: millisec
    Source: f5ca7481da8c038325d15612bb1c6473ce1ef16f
    Metric: kernel.all.cpu.nice
    labels {"agent":"linux","domainname":"localdomain",\
            "groupid":1000,"hostname":"shard",\
            "latitude":-25.28496,"longitude":152.87886,\
            "machineid":"295b16e3b6074cc8bdbda8bf96f6930a",\
            "userid":1000}
```

The complete set of **pmseries** metadata reporting options are:

options	Description
-a , **–all**	Convenience option to report all metadata for the given timeseries, equivalent to **-dilms**.
-d , **–desc**	Metric descriptions detailing the PMID, data type, data semantics, units, scale and associated instance domain. This option has a direct pminfo(1) equivalent.
-g *pattern* , **–glob** = *pattern*	Provide a glob(7) pattern to restrict the report provided by the **-i** , **-l** , **-m** and **-S**.
-i , **–instances**	Metric descriptions detailing the PMID, data type, data semantics, units, scale and associated instance domain.
-I , **–fullindom**	Print the InDom in verbose mode. This option has a direct pminfo(1) equivalent.
-l , **–labels**	Print label sets associated with metrics and instances. Labels are optional metric metadata described in detail in pmLookupLabels(3). This option has a direct pminfo(1) equivalent.
-m , **–metrics**	Print metric names.
-M , **–fullpmid**	Print the PMID in verbose mode. This option has a direct pminfo(1) equivalent.
-n , **–names**	Print comma-separated label names only (not values) for the labels associated with metrics and instances.
-s , **–series**	Print timeseries identifiers associated with metrics, instances and sources. These unique identifiers are calculated from intrinsic (non-optional) labels and other metric metadata associated with each PMAPI context (sources), metrics and instances. Archive, local context or pmcd(1) connections for the same host all produce the same source identifier.

References : pminfo(1) , glob(7) , pmLookupLabels(3) , pmcd(1)

11.6.2 Timeseries Sources

A source is a unique identifier (represented externally as a 40-byte hexadecimal SHA-1 hash) that represents both the live host and/or archives from which each timeseries originated. The context for a source identifier (obtained with **-s**) can be reported with:

-S , **–sources** : Print names for timeseries sources. These names are either hostnames or fully qualified archive paths.

It is important to note that live and archived sources can and will generate the same SHA-1 source identifier hash, provided that the context labels remain the same for that host (labels are stored in PCP archives and can also be fetched live from pmcd(1)).

11.6.3 Timeseries Loading

Timeseries metadata and data are loaded either automatically by a local pmproxy(1), or manually using a specially crafted **pmseries** query and the **-L/ –load** option:

```
$ pmseries --load "{source.path: \"$PCP_LOG_DIR/pmlogger/acme\"}"
pmseries: [Info] processed 2275 archive records from [...]
```

This query must specify a source archive path, but can also restrict the import to specific timeseries (using metric names, labels, etc) and to a specific time window using the time specification component of the query language.

As a convenience, if the argument to load is a valid file path as determined by access(2), then a short-hand form can be used:

```
$ pmseries --load $PCP_LOG_DIR/pmlogger/acme.0
```

11.6.4 Options

The available command line options, in addition to timeseries metadata and sources options described above, are:

options	Description
-c *config* , **–config** = *config*	Specify the *config* file to use.
-h *host* , **–host** = *host*	Connect Redis server at *host*, rather than the one the localhost.
-L , **–load**	Load timeseries metadata and data into the Redis cluster.
-p *port* , **–port** = *port*	Connect Redis server at *port*, rather than the default **6379** .
-q , **–query**	Perform a timeseries query. This is the default action.
-t , **–times**	Report time stamps numerically (in milliseconds) instead of the default human readable form.
-v , **–values**	Report all of the known values for given *label* name(s).
-V , **–version**	Display version number and exit.
-Z *timezone* , **–timezone** = *timezone*	Use timezone for the date and time. Timezone is in the format of the environment variable TZ as described in environ(7).
-? , **–help**	Display usage message and exit.

11.6.5 Examples

The following sample query shows several fundamental aspects of the **pmseries** query language:

```
$ pmseries 'kernel.all.load{hostname:"toium"}[count:2]'

eb713a9cf472f775aa59ae90c43cd7f960f7870f
    [Thu Nov 14 05:57:06.082861000 2019] 1.0e-01␣
↪b84040ffccd54f839b65140cf139bab51cbbcf62
    [Thu Nov 14 05:57:06.082861000 2019] 6.8e-01␣
↪a60b5b3bf25e71071c41934fa4d7d251f765f30c
    [Thu Nov 14 05:57:06.082861000 2019] 6.4e-01␣
↪e1974a062375e6e62370ffadf5b0650dad739480
    [Thu Nov 14 05:57:16.091546000 2019] 1.6e-01␣
↪b84040ffccd54f839b65140cf139bab51cbbcf62
    [Thu Nov 14 05:57:16.091546000 2019] 6.7e-01␣
↪a60b5b3bf25e71071c41934fa4d7d251f765f30c
    [Thu Nov 14 05:57:16.091546000 2019] 6.4e-01␣
↪e1974a062375e6e62370ffadf5b0650dad739480
```

This query returns the two most recent values for all instances of the **kernel.all.load** metric with a *label.hostname* matching the regular expression "toium". This is a set-valued metric (i.e., a metric with an "instance domain" which in this case consists of three instances: 1, 5 and 15 minute averages). The first column returned is a timestamp, then a floating point value, and finally an instance identifier timeseries hash (two values returned for three instances, so six rows are returned). The metadata for these timeseries can then be further examined:

```
$ pmseries -a eb713a9cf472f775aa59ae90c43cd7f960f7870f

eb713a9cf472f775aa59ae90c43cd7f960f7870f
    PMID: 60.2.0
    Data Type: float  InDom: 60.2 0xf000002
    Semantics: instant  Units: none
    Source: 0e89c1192db79326900d82131c31399524f0b3ee
    Metric: kernel.all.load
    inst [1 or "1 minute"] series b84040ffccd54f839b65140cf139bab51cbbcf62
    inst [5 or "5 minute"] series a60b5b3bf25e71071c41934fa4d7d251f765f30c
    inst [15 or "15 minute"] series e1974a062375e6e62370ffadf5b0650dad739480
    inst [1 or "1 minute"] labels {"agent":"linux","hostname":"toium"}
    inst [5 or "5 minute"] labels {"agent":"linux","hostname":"toium"}
    inst [15 or "15 minute"] labels {"agent":"linux","hostname":"toium"}
```

11.7 PCP Environment

Environment variables with the prefix **PCP_** are used to parameterize the file and directory names used by PCP. On each installation, the file */etc/pcp.conf* contains the local values for these variables. The `$PCP_CONF` variable may be used to specify an alternative configuration file, as described in pcp.conf(5).

For environment variables affecting PCP tools, see pmGetOptions(3).

11.8 PCP Grafana Plugin

The PCP Redis Grafana datasource from the PCP Grafana plugin queries the fast, scalable time series capabilities provided by the **pmseries** functionality. It is intended to query historical data across multiple hosts and supports filtering based on labels. This data source also provides a native interface between Grafana and Performance Co-Pilot (PCP), allowing PCP metric data to be presented in Grafana panels, such as graphs, tables, heatmaps, etc. Under the hood, the data source makes REST API query requests to the PCP pmproxy(1) service, which can be running either locally or on a remote host. The pmproxy daemon can be local or remote and uses the Redis time-series database (local or remote) for persistent storage.

For more information on PCP Grafana Plugin, visit PCP Grafana Plugin Documentation .

CHAPTER

TWELVE

PROGRAMMING PERFORMANCE CO-PILOT

Contents

- *Programming Performance Co-Pilot*
  - *PCP Architecture*
    - *Distributed Collection*
    - *Name Space*
    - *Distributed PMNS*
    - *Retrospective Sources of Performance Metrics*
  - *Overview of Component Software*
    - *Application and Agent Development*
  - *PMDA Development*
    - *Overview*
    - *Building a PMDA*
      - *In-Process (DSO) Method*
      - *Daemon Process Method*
  - *Client Development and PMAPI*
  - *Library Reentrancy and Threaded Applications*

Performance Co-Pilot (PCP) provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance management services. PCP is designed for the in-depth analysis and sophisticated control that are needed to understand and manage the hardest performance problems in the most complex systems.

PCP provides unparalleled power to quickly isolate and understand performance behavior, resource utilization, activity levels and performance bottlenecks.

Performance data may be collected and exported from multiple sources, most notably the hardware platform, the operating system kernel, layered services, and end-user applications.

There are several ways to extend PCP by programming certain of its components:

- By writing a Performance Metrics Domain Agent (PMDA) to collect performance metrics from an uncharted performance domain (Chapter 2, *Writing A PMDA*)
- By creating new analysis or visualization tools using documented functions from the Performance Metrics Application Programming Interface (PMAPI) (Chapter 3, *PMAPI–The Performance Metrics API*)

- By adding performance instrumentation to an application using facilities from PCP libraries, which offer both sampling and event tracing models.

Finally, the topic of customizing an installation is covered in the chapter on customizing and extending PCP service in the *Performance Co-Pilot User's and Administrator's Guide*.

12.1 PCP Architecture

This section gives a brief overview of PCP architecture.

PCP consists of numerous monitoring and collecting tools. **Monitoring tools** such as **pmval** and **pminfo** report on metrics, but have minimal interaction with target systems. **Collection tools**, called PMDAs, extract performance values from target systems, but do not provide user interfaces.

Systems supporting PCP services are broadly classified into two categories:

1. Collector: Hosts that have the PMCD and one or more PMDAs running to collect and export performance metrics
2. Monitor: Hosts that import performance metrics from one or more collector hosts to be consumed by tools to monitor, manage, or record the performance of the collector hosts

Each PCP enabled host can operate as a collector, or a monitor, or both.

Figure 1.1. PCP Global Process Architecture shows the architecture of PCP. The monitoring tools consume and process performance data using a public interface, the Performance Metrics Application Programming Interface (PMAPI).

Below the PMAPI level is the PMCD process, which acts in a coordinating role, accepting requests from clients, routing requests to one or more PMDAs, aggregating responses from the PMDAs, and responding to the requesting client.

Each performance metric domain (such as the operating system kernel or a database management system) has a well-defined name space for referring to the specific performance metrics it knows how to collect.

Fig. 1: Figure 1.1. PCP Global Process Architecture

12.1.1 Distributed Collection

The performance metrics collection architecture is distributed, in the sense that any monitoring tool may be executing remotely. However, a PMDA is expected to be running on the operating system for which it is collecting performance measurements; there are some notable PMDAs such as Cisco and Cluster that are exceptions, and collect performance data from remote systems.

As shown in *Figure 1.2. Process Structure for Distributed Operation*, monitoring tools communicate only with PMCD. The PMDAs are controlled by PMCD and respond to requests from the monitoring tools that are forwarded by PMCD to the relevant PMDAs on the collector host.

Fig. 2: Figure 1.2. Process Structure for Distributed Operation

The host running the monitoring tools does not require any collection tools, including PMCD, since all requests for metrics are sent to the PMCD process on the collector host.

The connections between monitoring tools and PMCD processes are managed in **libpcp**, below the PMAPI level; see the **PMAPI(3)** man page. Connections between PMDAs and PMCD are managed by the PMDA functions; see the

PMDA(3) and **pmcd(1)** man pages. There can be multiple monitor clients and multiple PMDAs on the one host, but there may be only one PMCD process.

12.1.2 Name Space

Each PMDA provides a domain of metrics, whether they be for the operating system, a database manager, a layered service, or an application module. These metrics are referred to by name inside the user interface, and with a numeric Performance Metric Identifier (PMID) within the underlying PMAPI.

The PMID consists of three fields: the domain, the cluster, and the item number of the metric. The domain is a unique number assigned to each PMDA. For example, two metrics with the same domain number must be from the same PMDA. The cluster and item numbers allow metrics to be easily organized into groups within the PMDA, and provide a hierarchical taxonomy to guarantee uniqueness within each PMDA.

The Performance Metrics Name Space (PMNS) describes the exported performance metrics, in particular the mapping from PMID to external name, and vice-versa.

12.1.3 Distributed PMNS

Performance metric namespace (PMNS) operations are directed by default to the host or set of archives that is the source of the desired performance metrics.

In *Figure 1.2. Process Structure for Distributed Operation*, both Performance Metrics Collection Daemon (PMCD) processes would respond to PMNS queries from monitoring tools by referring to their local PMNS. If different PMDAs were installed on the two hosts, then the PMNS used by each PMCD would be different, to reflect variations in available metrics on the two hosts.

Although extremely rarely used, the **-n** *pmnsfile* command line option may be used with many PCP monitoring tools to force use of a local PMNS file in preference to the PMNS at the source of the metrics.

12.1.4 Retrospective Sources of Performance Metrics

The distributed collection architecture described in the previous section is used when PMAPI clients are requesting performance metrics from a real-time or live source.

The PMAPI also supports delivery of performance metrics from a historical source in the form of a PCP archive log. Archive logs are created using the **pmlogger** utility, and are replayed in an architecture as shown in *Figure 1.3. Architecture for Retrospective Analysis*.

Fig. 3: Figure 1.3. Architecture for Retrospective Analysis

12.2 Overview of Component Software

Performance Co-Pilot (PCP) is composed of text-based tools, optional graphical tools, and related commands. Each tool or command is fully documented by a man page. These man pages are named after the tools or commands they describe, and are accessible through the **man** command. For example, to see the **pminfo(1)** man page for the **pminfo** command, enter this command:

```
man pminfo
```

A list of PCP developer tools and commands, grouped by functionality, is provided in the following section.

12.2.1 Application and Agent Development

The following PCP tools aid the development of new programs to consume performance data, and new agents to export performance data within the PCP framework:

chkhelp

Checks the consistency of performance metrics help database files.

dbpmda

Allows PMDA behavior to be exercised and tested. It is an interactive debugger for PMDAs.

mmv

Is used to instrument applications using Memory Mapped Values (MMV). These are values that are communicated with pmcd instantly, and very efficiently, using a shared memory mapping. It is a program instrumentation library.

newhelp

Generates the database files for one or more source files of PCP help text.

pmapi

Defines a procedural interface for developing PCP client applications. It is the Performance Metrics Application Programming Interface (PMAPI).

pmclient

Is a simple client that uses the PMAPI to report some high-level system performance metrics. The source code for **pmclient** is included in the distribution.

pmda

Is a library used by many shipped PMDAs to communicate with a **pmcd** process. It can expedite the development of new and custom PMDAs.

pmgenmap

Generates C declarations and **cpp** macros to aid the development of customized programs that use the facilities of PCP. It is a program development tool.

12.3 PMDA Development

A collection of Performance Metrics Domain Agents (PMDAs) are provided with PCP to extract performance metrics. Each PMDA encapsulates domain-specific knowledge and methods about performance metrics that implement the uniform access protocols and functional semantics of the PCP. There is one PMDA for the operating system, another for process specific statistics, one each for common DBMS products, and so on. Thus, the range of performance metrics can be easily extended by implementing and integrating new PMDAs. Chapter 2, *Writing A PMDA*, is a step-by-step guide to writing your own PMDA.

12.3.1 Overview

Once you are familiar with the PCP and PMDA frameworks, you can quickly implement a new PMDA with only a few data structures and functions. This book contains detailed discussions of PMDA architecture and the integration of PMDAs into the PCP framework. This includes integration with PMCD. However, details of extracting performance metrics from the underlying instrumentation vary from one domain to another and are not covered in this book.

A PMDA is responsible for a set of performance metrics, in the sense that it must respond to requests from PMCD for information about performance metrics, instance domains, and instantiated values. The PMCD process generates requests on behalf of monitoring tools that make requests using PMAPI functions.

You can incorporate new performance metrics into the PCP framework by creating a PMDA, then reconfiguring PMCD to communicate with the new PMDA.

12.3.2 Building a PMDA

A PMDA interacts with PMCD across one of several well-defined interfaces and protocol mechanisms. These implementation options are described in the *Performance Co-Pilot User's and Administrator's Guide*.

Note: It is strongly recommended that code for a new PMDA be based on the source of one of the existing PMDAs below the `${PCP_PMDAS_DIR}` directory.

In-Process (DSO) Method

This method of building a PMDA uses a Dynamic Shared Object (DSO) that is attached by PMCD, using the platform-specific shared library manipulation interfaces such as **dlopen(3)**, at initialization time. This is the highest performance option (there is no context switching and no interprocess communication (IPC) between the PMCD and the PMDA), but is operationally intractable in some situations. For example, difficulties arise where special access permissions are required to read the instrumentation behind the performance metrics (**pmcd** does not run as root), or where the performance metrics are provided by an existing process with a different protocol interface. The DSO PMDA effectively executes as part of PMCD; so great care is required when crafting a PMDA in this manner. Calls to **exit(1)** in the PMDA, or a library it uses, would cause PMCD to exit and end monitoring of that host. Other implications are discussed in Section 2.2.3, "*Daemon PMDA*".

Daemon Process Method

Functionally, this method may be thought of as a DSO implementation with a standard **main** routine conversion wrapper so that communication with PMCD uses message passing rather than direct procedure calls. For some very basic examples, see the `${PCP_PMDAS_DIR}/trivial/trivial.c` and `${PCP_PMDAS_DIR}/simple/simple.c` source files.

The daemon PMDA is actually the most common, because it allows multiple threads of control, greater (different user) privileges when executing, and provides more resilient error encapsulation than the DSO method.

Note: Of particular interest for daemon PMDA writers, the `${PCP_PMDAS_DIR}/simple` PMDA has implementations in C, Perl and Python.

12.4 Client Development and PMAPI

Application developers are encouraged to create new PCP client applications to monitor, display, and analyze performance data in a manner suited to their particular site, application suite, or information processing environment.

PCP client applications are programmed using the Performance Metrics Application Programming Interface (PMAPI), documented in Chapter 3, *PMAPI–The Performance Metrics API*. The PMAPI, which provides performance tool developers with access to all of the historical and live distributed services of PCP, is the interface used by the standard PCP utilities.

12.5 Library Reentrancy and Threaded Applications

While the core PCP library (**libpcp**) is thread safe, the layered PMDA library (**libpcp_pmda**) is not. This is a deliberate design decision to trade-off commonly required performance and efficiency against the less common requirement for multiple threads of control to call the PCP libraries.

The simplest and safest programming model is to designate at most one thread to make calls into the PCP PMDA library.

CHAPTER

THIRTEEN

WRITING A PMDA

Contents

- *Writing A PMDA*
  - *Implementing a PMDA*
  - *PMDA Architecture*
    - *Overview*
    - *DSO PMDA*
    - *Daemon PMDA*
    - *Caching PMDA*
  - *Domains, Metrics, Instances and Labels*
    - *Overview*
    - *Domains*
    - *Metrics*
      - *Data Structures*
      - *Semantics*
    - *Instances*
      - *Instance Identification*
      - *N Dimensional Data*
      - *Data Structures*
    - *Labels*
      - *Label Hierarchy*
      - *Data Structures*
  - *Other Issues*
    - *Extracting the Information*
    - *Latency and Threads of Control*
    - *Name Space*
    - *PMDA Help Text*

        * *Management of Evolution within a PMDA*
    - *PMDA Interface*
        * *Overview*
            · *Trivial PMDA*
            · *Simple PMDA*
            · *simple_store in the Simple PMDA*
            · *Return Codes for pmdaFetch Callbacks*
        * *PMDA Structures*
    - *Initializing a PMDA*
        * *Overview*
        * *Common Initialization*
            · *Trivial PMDA*
            · *Simple PMDA*
        * *Daemon Initialization*
    - *Testing and Debugging a PMDA*
        * *Overview*
        * *Debugging Information*
        * *dbpmda Debug Utility*
    - *Integration of a PMDA*
        * *Installing a PMDA*
        * *Removing a PMDA*
        * *Configuring PCP Tools*

This chapter constitutes a programmer's guide to writing a Performance Metrics Domain Agent (PMDA) for Performance Co-Pilot (PCP).

The presentation assumes the developer is using the standard PCP **libpcp_pmda** library, as documented in the **PMDA(3)** and associated man pages.

13.1 Implementing a PMDA

The job of a PMDA is to gather performance data and report them to the Performance Metrics Collection Daemon (PMCD) in response to requests from PCP monitoring tools routed to the PMDA via PMCD.

An important requirement for any PMDA is that it have low latency response to requests from PMCD. Either the PMDA must use a quick access method and a single thread of control, or it must have asynchronous refresh and two threads of control: one for communicating with PMCD, the other for updating the performance data.

The PMDA is typically acting as a gateway between the target domain (that is, the performance instrumentation in an application program or service) and the PCP framework. The PMDA may extract the information using one of a number of possible export options that include a shared memory segment or **mmap** file; a sequential log file (where the PMDA parses the tail of the log file to extract the information); a snapshot file (the PMDA rereads the file as required); or application-specific communication services (IPC).

Note: The choice of export methodology is typically determined by the source of the instrumentation (the target domain) rather than by the PMDA.

Procedure 2.1. Creating a PMDA describes the suggested steps for designing and implementing a PMDA:

Procedure 2.1. Creating a PMDA

1. Determine how to extract the metrics from the target domain.
2. Select an appropriate architecture for the PMDA (daemon or DSO, IPC, **pthreads** or single threaded).
3. Define the metrics and instances that the PMDA will support.
4. Implement the functionality to extract the metric values.
5. Assign Performance Metric Identifiers (PMIDs) for the metrics, along with names for the metrics in the Performance Metrics Name Space (PMNS). These concepts will be further expanded in Section 2.3, "*Domains, Metrics, Instances and Labels*"
6. Specify the help file and control data structures for metrics and instances that are required by the standard PMDA implementation library functions.
7. Write code to supply the metrics and associated information to PMCD.
8. Implement any PMDA-specific callbacks, and PMDA initialization functions.
9. Exercise and test the PMDA with the purpose-built PMDA debugger; see the **dbpmda(1)** man page.
10. Install and connect the PMDA to a running PMCD process; see the **pmcd(1)** man page.
11. Where appropriate, define **pmie** rule templates suitable for alerting or notification systems. For more information, see the **pmie(1)** and **pmieconf(1)** man pages.
12. Where appropriate, define **pmlogger** configuration templates suitable for creating PCP archives containing the new metrics. For more information, see the **pmlogconf(1)** and **pmlogger(1)** man pages.

13.2 PMDA Architecture

This section discusses the two methods of connecting a PMDA to a PMCD process:

1. As a separate process using some interprocess communication (IPC) protocol.
2. As a dynamically attached library (that is, a dynamic shared object or DSO).

13.2.1 Overview

All PMDAs are launched and controlled by the PMCD process on the local host. PMCD receives requests from the monitoring tools and forwards them to the PMDAs. Responses, when required, are returned through PMCD to the clients. The requests fall into a small number of categories, and the PMDA must handle each request type. For a DSO PMDA, each request type corresponds to a method in the agent. For a daemon PMDA, each request translates to a message or protocol data unit (PDU) that may be sent to a PMDA from PMCD.

For a daemon PMDA, the following request PDUs must be supported:

PDU_FETCH

Request for metric values (see the **pmFetch(3)** man page.)

PDU_PROFILE

A list of instances required for the corresponding metrics in subsequent fetches (see the **pmAddProfile(3)** man page).

PDU_INSTANCE_REQ

Request for a particular instance domain for instance descriptions (see the **pmGetInDom(3)** man page).

PDU_DESC_REQ

Request for metadata describing metrics (see the **pmLookupDesc(3)** man page).

PDU_TEXT_REQ

Request for metric help text (see the **pmLookupText(3)** man page).

PDU_RESULT

Values to store into metrics (see the **pmStore(3)** man page).

The following request PDUs may optionally be supported:

PDU_PMNS_NAMES

Request for metric names, given one or more identifiers (see the **pmLookupName(3)** man page.)

PDU_PMNS_CHILD

A list of immediate descendent nodes of a given namespace node (see the **pmGetChildren(3)** man page).

PDU_PMNS_TRAVERSE

Request for a particular sub-tree of a given namespace node (see the **pmTraversePMNS(3)** man page).

PDU_PMNS_IDS

Perform a reverse name lookup, mapping a metric identifier to a name (see the **pmNameID(3)** man page).

PDU_ATTR

Handle connection attributes (key/value pairs), such as client credentials and other authentication information (see the **__pmParseHostAttrsSpec(3)** man page).

PDU_LABEL_REQ

Request for metric labels (see the **pmLookupLabels(3)** man page).

Each PMDA is associated with a unique domain number that is encoded in the domain field of metric and instance identifiers, and PMCD uses the domain number to determine which PMDA can handle the components of any given client request.

13.2.2 DSO PMDA

Each PMDA is required to implement a function that handles each of the request types. By implementing these functions as library functions, a PMDA can be implemented as a dynamically shared object (DSO) and attached by PMCD at run time with a platform-specific call, such as **dlopen**; see the **dlopen(3)** man page. This eliminates the need for an IPC layer (typically a pipe) between each PMDA and PMCD, because each request becomes a function call rather than a message exchange. The required library functions are detailed in Section 2.5, "*PMDA Interface*".

A PMDA that interacts with PMCD in this fashion must abide by a formal initialization protocol so that PMCD can discover the location of the library functions that are subsequently called with function pointers. When a DSO PMDA is installed, the PMCD configuration file, `${PCP_PMCDCONF_PATH}`, is updated to reflect the domain and name of the PMDA, the location of the shared object, and the name of the initialization function. The initialization sequence is discussed in Section 2.6, "*Initializing a PMDA*".

As superuser, install the simple PMDA as a DSO, as shown in *Example 2.1. Simple PMDA as a DSO*, and observe the changes in the PMCD configuration file. The output may differ slightly depending on the operating system you are using, any other PMDAs you have installed or any PMCD access controls you have in place.

Example 2.1. Simple PMDA as a DSO

```
# cat ${PCP_PMCDCONF_PATH}
# Performance Metrics Domain Specifications
#
# This file is automatically generated during the build
# Name  Id      IPC     IPC Params      File/Cmd
root    1       pipe    binary          /var/lib/pcp/pmdas/root/pmdaroot
pmcd    2       dso     pmcd_init       ${PCP_PMDAS_DIR}/pmcd/pmda_pmcd.so
proc    3       pipe    binary          ${PCP_PMDAS_DIR}/linux/pmda_proc.so -d 3
linux   60      dso     linux_init      ${PCP_PMDAS_DIR}/linux/pmda_linux.so
mmv     70      dso     mmv_init        /var/lib/pcp/pmdas/mmv/pmda_mmv.so
simple  254     dso     simple_init     ${PCP_PMDAS_DIR}/simple/pmda_simple.so
```

As can be seen from the contents of `${PCP_PMCDCONF_PATH}`, the DSO version of the simple PMDA is in a library named **pmda_simple.so** and has an initialization function called **simple_init**. The domain of the simple PMDA is 254, as shown in the column headed **Id**.

Note: For some platforms the DSO file name will not be **pmda_simple.so**. On Mac OS X it is **pmda_simple.dylib** and on Windows it is **pmda_simple.dll**.

13.2.3 Daemon PMDA

A DSO PMDA provides the most efficient communication between the PMDA and PMCD. This approach has some disadvantages resulting from the DSO PMDA being the same process as PMCD:

- An error or bug that causes a DSO PMDA to exit also causes PMCD to exit, which affects all connected client tools.
- There is only one thread of control in PMCD; as a result, a computationally expensive PMDA, or worse, a PMDA that blocks for I/O, adversely affects the performance of PMCD.
- PMCD runs as the "pcp" user; so all DSO PMDAs must also run as this user.
- A memory leak in a DSO PMDA also causes a memory leak for PMCD.

Consequently, many PMDAs are implemented as a daemon process.

The **libpcp_pmda** library is designed to allow simple implementation of a PMDA that runs as a separate process. The library functions provide a message passing layer acting as a generic wrapper that accepts PDUs, makes library calls using the standard DSO PMDA interface, and sends PDUs. Therefore, you can implement a PMDA as a DSO and then install it as either a daemon or a DSO, depending on the presence or absence of the generic wrapper.

The PMCD process launches a daemon PMDA with **fork** and **execv** (or **CreateProcess** on Windows). You can easily connect a pipe to the PMDA using standard input and output. The PMCD process may also connect to a daemon PMDA using IPv4 or IPv6 TCP/IP, or UNIX domain sockets if the platform supports that; see the **tcp(7), ip(7), ipv6(7)** or **unix(7)** man pages.

As superuser, install the simple PMDA as a daemon process as shown in *Example 2.2. Simple PMDA as a Daemon*. Again, the output may differ due to operating system differences, other PMDAs already installed, or access control sections in the PMCD configuration file.

Example 2.2. Simple PMDA as a Daemon

The specification for the simple PMDA now states the connection type of **pipe** to PMCD and the executable image for the PMDA is `${PCP_PMDAS_DIR}/simple/pmdasimple`, using domain number 253.

```
# cd ${PCP_PMDAS_DIR}/simple
# ./Install
...
Install simple as a daemon or dso agent? [daemon] daemon
PMCD should communicate with the daemon via pipe or socket? [pipe] pipe
...
# cat ${PCP_PMCDCONF_PATH}
# Performance Metrics Domain Specifications
#
# This file is automatically generated during the build
# Name  Id      IPC     IPC Params      File/Cmd
root    1       pipe    binary          /var/lib/pcp/pmdas/root/pmdaroot
pmcd    2       dso     pmcd_init       ${PCP_PMDAS_DIR}/pmcd/pmda_pmcd.so
proc    3       pipe    binary          ${PCP_PMDAS_DIR}/linux/pmda_proc.so -d 3
linux   60      dso     linux_init      ${PCP_PMDAS_DIR}/linux/pmda_linux.so
mmv     70      dso     mmv_init        /var/lib/pcp/pmdas/mmv/pmda_mmv.so
simple  253     pipe    binary          ${PCP_PMDAS_DIR}/simple/pmdasimple -d 253
```

13.2.4 Caching PMDA

When either the cost or latency associated with collecting performance metrics is high, the PMDA implementer may choose to trade off the currency of the performance data to reduce the PMDA resource demands or the fetch latency time.

One scheme for doing this is called a caching PMDA, which periodically instantiates values for the performance metrics and responds to each request from PMCD with the most recently instantiated (or cached) values, as opposed to instantiating current values on demand when the PMCD asks for them.

The Cisco PMDA is an example of a caching PMDA. For additional information, see the contents of the `${PCP_PMDAS_DIR}/cisco` directory and the **pmdacisco(1)** man page.

13.3 Domains, Metrics, Instances and Labels

This section defines metrics and instances, discusses how they should be designed for a particular target domain, and shows how to implement support for them.

The examples in this section are drawn from the trivial and simple PMDAs. Refer to the `${PCP_PMDAS_DIR}/trivial` and `${PCP_PMDAS_DIR}/simple` directories, respectively, where both binaries and source code are available.

13.3.1 Overview

Domains are autonomous performance areas, such as the operating system or a layered service or a particular application. *Metrics* are raw performance data for a domain, and typically quantify activity levels, resource utilization or quality of service. *Instances* are sets of related metrics, as for multiple processors, or multiple service classes, or multiple transaction types.

PCP employs the following simple and uniform data model to accommodate the demands of performance metrics drawn from multiple domains:

- Each metric has an identifier that is unique across all metrics for all PMDAs on a particular host.

- Externally, metrics are assigned names for user convenience–typically there is a 1:1 relationship between a metric name and a metric identifier.
- The PMDA implementation determines if a particular metric has a singular value or a set of (zero or more) values. For instance, the metric **hinv.ndisk** counts the number of disks and has only one value on a host, whereas the metric **disk.dev.total** counts disk I/O operations and has one value for each disk on the host.
- If a metric has a set of values, then members of the set are differentiated by instances. The set of instances associated with a metric is an *instance domain*. For example, the set of metrics **disk.dev.total** is defined over an instance domain that has one member per disk spindle.

The selection of metrics and instances is an important design decision for a PMDA implementer. The metrics and instances for a target domain should have the following qualities:

- Obvious to a user
- Consistent across the domain
- Accurately representative of the operational and functional aspects of the domain

For each metric, you should also consider these questions:

- How useful is this value?
- What units give a good sense of scale?
- What name gives a good description of the metric's meaning?
- Can this metric be combined with another to convey the same useful information?

As with all programming tasks, expect to refine the choice of metrics and instances several times during the development of the PMDA.

13.3.2 Domains

Each PMDA must be uniquely identified by PMCD so that requests from clients can be efficiently routed to the appropriate PMDA. The unique identifier, the PMDA's domain, is encoded within the metrics and instance domain identifiers so that they are associated with the correct PMDA, and so that they are unique, regardless of the number of PMDAs that are connected to the PMCD process.

The default domain number for each PMDA is defined in `${PCP_VAR_DIR}/pmns/stdpmid`. This file is a simple table of PMDA names and their corresponding domain number. However, a PMDA does not have to use this domain number–the file is only a guide to help avoid domain number clashes when PMDAs are installed and activated.

The domain number a PMDA uses is passed to the PMDA by PMCD when the PMDA is launched. Therefore, any data structures that require the PMDA's domain number must be set up when the PMDA is initialized, rather than declared statically. The protocol for PMDA initialization provides a standard way for a PMDA to implement this run-time initialization.

Note: Although uniqueness of the domain number in the `${PCP_PMCDCONF_PATH}` control file used by PMCD is all that is required for successful starting of PMCD and the associated PMDAs, the developer of a new PMDA is encouraged to add the default domain number for each new PMDA to the `${PCP_VAR_DIR}/pmns/stdpmid.local` file and then to run the **Make.stdpmid** script in `${PCP_VAR_DIR}/pmns` to recreate `${PCP_VAR_DIR}/pmns/stdpmid`; this file acts as a repository for documenting the known default domain numbers.

13.3.3 Metrics

A PMDA provides support for a collection of metrics. In addition to the obvious performance metrics, and the measures of time, activity and resource utilization, the metrics should also describe how the target domain has been configured, as this can greatly affect the correct interpretation of the observed performance. For example, metrics that describe network transfer rates should also describe the number and type of network interfaces connected to the host (**hinv.ninterface**, **network.interface.speed, network.interface.duplex**, and so on)

In addition, the metrics should describe how the PMDA has been configured. For example, if the PMDA was periodically probing a system to measure quality of service, there should be metrics for the delay between probes, the number of probes attempted, plus probe success and failure counters. It may also be appropriate to allow values to be stored (see the **pmstore(1)** man page) into the delay metric, so that the delay used by the PMDA can be altered dynamically.

Data Structures

Each metric must be described in a **pmDesc** structure; see the **pmLookupDesc(3)** man page:

```
typedef struct {
    pmID        pmid;           /* unique identifier */
    int         type;           /* base data type */
    pmInDom     indom;          /* instance domain */
    int         sem;            /* semantics of value */
    pmUnits     units;          /* dimension and units */
} pmDesc;
```

This structure contains the following fields:

pmid

A unique identifier, Performance Metric Identifier (PMID), that differentiates this metric from other metrics across the union of all PMDAs

type

A data type indicator showing whether the format is an integer (32 or 64 bit, signed or unsigned); float; double; string; or arbitrary aggregate of binary data

indom

An instance domain identifier that links this metric to an instance domain

sem

An encoding of the value's semantics (counter, instantaneous, or discrete)

units

A description of the value's units based on dimension and scale in the three orthogonal dimensions of space, time, and count (or events)

Note: This information can be observed for metrics from any active PMDA using **pminfo** command line options, for example:

```
$ pminfo -d -m network.interface.out.drops

network.interface.out.drops PMID: 60.3.11
    Data Type: 64-bit unsigned int  InDom: 60.3 0xf000003
    Semantics: counter  Units: count
```

Symbolic constants of the form **PM_TYPE*, PM_SEM_*, PM_SPACE_*, PM_TIME_***, and **PM_COUNT_*** are defined in the **<pcp/pmapi.h>** header file. You may use them to initialize the elements of a **pmDesc** structure. The **pmID** type is an unsigned integer that can be safely cast to a **__pmID_int** structure, which contains fields defining the metric's (PMDA's) domain, cluster, and item number as shown in *Example 2.3. __pmID_int Structure*:

Example 2.3. __pmID_int Structure

```
typedef struct {
        int             flag:1;
        unsigned int    domain:9;
        unsigned int    cluster:12;
        unsigned int    item:10;
} __pmID_int;
```

For additional information, see the **<pcp/libpcp.h>** file.

The **flag** field should be ignored. The **domain** number should be set at run time when the PMDA is initialized. The **PMDA_PMID** macro defined in **<pcp/pmapi.h>** can be used to set the **cluster** and **item** fields at compile time, as these should always be known and fixed for a particular metric.

Note: The three components of the PMID should correspond exactly to the three-part definition of the PMID for the corresponding metric in the PMNS described in Section 2.4.3, "*Name Space*".

A table of **pmdaMetric** structures should be defined within the PMDA, with one structure per metric as shown in *Example 2.4. pmdaMetric Structure*.

Example 2.4. pmdaMetric Structure

```
typedef struct {
    void        *m_user;        /* for users external use */
    pmDesc      m_desc;         /* metric description */
} pmdaMetric;
```

This structure contains a **pmDesc** structure and a handle that allows PMDA-specific structures to be associated with each metric. For example, **m_user** could be a pointer to a global variable containing the metric value, or a pointer to a function that may be called to instantiate the metric's value.

The trivial PMDA, shown in *Example 2.5. Trivial PMDA*, has only a singular metric (that is, no instance domain):

Example 2.5. Trivial PMDA

```
static pmdaMetric metrictab[] = {
/* time */
 { NULL,
   { PMDA_PMID(0, 1), PM_TYPE_U32, PM_INDOM_NULL, PM_SEM_INSTANT,
     PMDA_PMUNITS(0, 1, 0, 0, PM_TIME_SEC, 0) }, },
};
```

This single metric (**trivial.time**) has the following:

- A PMID with a cluster of 0 and an item of 1. Note that this is not yet a complete PMID, the domain number which identifies the PMDA will be combined with it at runtime.
- An unsigned 32-bit integer (**PM_TYPE_U32**)
- A singular value and hence no instance domain (**PM_INDOM_NULL**)
- An instantaneous semantic value (**PM_SEM_INSTANT**)

- Dimension “time” and the units “seconds”

Semantics

The metric’s semantics describe how PCP tools should interpret the metric’s value. The following are the possible semantic types:

- Counter (**PM_SEM_COUNTER**)
- Instantaneous value (**PM_SEM_INSTANT**)
- Discrete value (**PM_SEM_DISCRETE**)

A counter should be a value that monotonically increases (or monotonically decreases, which is less likely) with respect to time, so that the rate of change should be used in preference to the actual value. Rate conversion is not appropriate for metrics with instantaneous values, as the value is a snapshot and there is no basis for assuming any values that might have been observed between snapshots. Discrete is similar to instantaneous; however, once observed it is presumed the value will persist for an extended period (for example, system configuration, static tuning parameters and most metrics with non-numeric values).

For a given time interval covering six consecutive timestamps, each spanning two units of time, the metric values in *Example 2.6. Effect of Semantics on a Metric* are exported from a PMDA (“N/A” implies no value is available):

Example 2.6. Effect of Semantics on a Metric

```
Timestamps:          1   3   5   7   9  11
Value:              10  30  60  80  90 N/A
```

The default display of the values would be as follows:

```
Timestamps:          1   3   5   7   9  11
Semantics:
Counter            N/A  10  15  10   5 N/A
Instantaneous       10  30  60  80  90 N/A
Discrete            10  30  60  80  90  90
```

Note that these interpretations of metric semantics are performed by the monitor tool, automatically, before displaying a value and they are not transformations that the PMDA performs.

13.3.4 Instances

Singular metrics have only one value and no associated instance domain. Some metrics contain a set of values that share a common set of semantics for a specific instance, such as one value per processor, or one value per disk spindle, and so on.

Note: The PMDA implementation is solely responsible for choosing the instance identifiers that differentiate instances within the instance domain. The PMDA is also responsible for ensuring the uniqueness of instance identifiers in any instance domain, as described in Section 2.3.4.1, “*Instance Identification*”.

Instance Identification

Consistent interpretation of instances and instance domains require a few simple rules to be followed by PMDA authors. The PMDA library provides a series of **pmdaCache** routines to assist.

- Each internal instance identifier (numeric) must be a unique 31-bit number.
- The external instance name (string) must be unique.
- When the instance name contains a space, the name to the left of the first space (the short name) must also be unique.
- Where an external instance name corresponds to some object or entity, there is an expectation that the association between the name and the object is fixed.
- It is preferable, although not mandatory, for the association between and external instance name (string) and internal instance identifier (numeric) to be persistent.

N Dimensional Data

Where the performance data can be represented as scalar values (singular metrics) or one-dimensional arrays or lists (metrics with an instance domain), the PCP framework is more than adequate. In the case of metrics with an instance domain, each array or list element is associated with an instance from the instance domain.

To represent two or more dimensional arrays, the coordinates must be one of the following:

- Mapped onto one dimensional coordinates.
- Enumerated into the Performance Metrics Name Space (PMNS).

For example, this 2 x 3 array of values called M can be represented as instances 1,. . . , 6 for a metric M:

```
M[1]   M[2]   M[3]
M[4]   M[5]   M[6]
```

Or they can be represented as instances 1, 2, 3 for metric M1 and instances 1, 2, 3 for metric M2:

```
M1[1]  M1[2]  M1[3]
M2[1]  M2[2]  M2[3]
```

The PMDA implementer must decide and consistently export this encoding from the N-dimensional instrumentation to the 1-dimensional data model of the PCP. The use of metric label metadata - arbitrary key/value pairs - allows the implementer to capture the higher dimensions of the performance data.

In certain special cases (for example, such as for a histogram), it may be appropriate to export an array of values as raw binary data (the type encoding in the descriptor is **PM_TYPE_AGGREGATE**). However, this requires the development of special PMAPI client tools, because the standard PCP tools have no knowledge of the structure and interpretation of the binary data. The usual issues of platform-dependence must also be kept in mind for this case - endianness, word-size, alignment and so on - the (possibly remote) special PMAPI client tools may need this information in order to decode the data successfully.

Data Structures

If the PMDA is required to support instance domains, then for each instance domain the unique internal instance identifier and external instance identifier should be defined using a **pmdaInstid** structure as shown in *Example 2.7. pmdaInstid Structure*:

Example 2.7. pmdaInstid Structure

```
typedef struct {
    int         i_inst;         /* internal instance identifier */
    char        *i_name;        /* external instance identifier */
} pmdaInstid;
```

The **i_inst** instance identifier must be a unique integer within a particular instance domain.

The complete instance domain description is specified in a **pmdaIndom** structure as shown in *Example 2.8. pmdaIndom Structure*:

Example 2.8. pmdaIndom Structure

```
typedef struct {
    pmInDom     it_indom;       /* indom, filled in */
    int         it_numinst;     /* number of instances */
    pmdaInstid  *it_set;        /* instance identifiers */
} pmdaIndom;
```

The **it_indom** element contains a **pmInDom** that must be unique across every PMDA. The other fields of the **pmdaIndom** structure are the number of instances in the instance domain and a pointer to an array of instance descriptions.

Example 2.9. __pmInDom_int Structure shows that the **pmInDom** can be safely cast to **__pmInDom_int**, which specifies the PMDA's domain and the instance number within the PMDA:

Example 2.9. __pmInDom_int Structure

```
typedef struct {
        int             flag:1;
        unsigned int    domain:9;   /* the administrative PMD */
        unsigned int    serial:22;  /* unique within PMD */
} __pmInDom_int;
```

As with metrics, the PMDA domain number is not necessarily known until run time; so the **domain** field must be set up when the PMDA is initialized.

For information about how an instance domain may also be associated with more than one metric, see the **pmdaInit(3)** man page.

The simple PMDA, shown in *Example 2.10. Simple PMDA*, has five metrics and two instance domains of three instances.

Example 2.10. Simple PMDA

```
/*
 * list of instances
 */
static pmdaInstid color[] = {
    { 0, "red" }, { 1, "green" }, { 2, "blue" }
};
static pmdaInstid       *timenow = NULL;
static unsigned int     timesize = 0;
/*
```

(continues on next page)

(continued from previous page)

```
 * list of instance domains
 */
static pmdaIndom indomtab[] = {
#define COLOR_INDOM     0
    { COLOR_INDOM, 3, color },
#define NOW_INDOM       1
   { NOW_INDOM, 0, NULL },
};
/*
 * all metrics supported in this PMDA - one table entry for each
 */
static pmdaMetric metrictab[] = {
/* numfetch */
    { NULL,
      { PMDA_PMID(0, 0), PM_TYPE_U32, PM_INDOM_NULL, PM_SEM_INSTANT,
       PMDA_PMUNITS(0, 0, 0, 0, 0, 0) }, },
/* color */
    { NULL,
      { PMDA_PMID(0, 1), PM_TYPE_32, COLOR_INDOM, PM_SEM_INSTANT,
        PMDA_PMUNITS(0, 0, 0, 0, 0, 0) }, },
/* time.user */
    { NULL,
      { PMDA_PMID(1, 2), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        PMDA_PMUNITS(0, 1, 0, 0, PM_TIME_SEC, 0) }, },
/* time.sys */
    { NULL,
      { PMDA_PMID(1,3), PM_TYPE_DOUBLE, PM_INDOM_NULL, PM_SEM_COUNTER,
        PMDA_PMUNITS(0, 1, 0, 0, PM_TIME_SEC, 0) }, },
/* now */
    { NULL,
      { PMDA_PMID(2,4), PM_TYPE_U32, NOW_INDOM, PM_SEM_INSTANT,
        PMDA_PMUNITS(0, 0, 0, 0, 0, 0) }, },
};
```

The metric **simple.color** is associated, via **COLOR_INDOM**, with the first instance domain listed in **indomtab**. PMDA initialization assigns the correct domain portion of the instance domain identifier in **indomtab[0].it_indom** and **metrictab[1].m_desc.indom**. This instance domain has three instances: red, green, and blue.

The metric **simple.now** is associated, via **NOW_INDOM**, with the second instance domain listed in **indomtab**. PMDA initialization assigns the correct domain portion of the instance domain identifier in **indomtab[1].it_indom** and **metrictab[4].m_desc.indom**. This instance domain is dynamic and initially has no instances.

All other metrics are singular, as specified by **PM_INDOM_NULL**.

In some cases an instance domain may vary dynamically after PMDA initialization (for example, **simple.now**), and this requires some refinement of the default functions and data structures of the **libpcp_pmda** library. Briefly, this involves providing new functions that act as wrappers for **pmdaInstance** and **pmdaFetch** while understanding the dynamics of the instance domain, and then overriding the instance and fetch methods in the **pmdaInterface** structure during PMDA initialization.

For the simple PMDA, the wrapper functions are **simple_fetch** and **simple_instance**, and defaults are over-ridden by the following assignments in the **simple_init** function:

```
dp->version.any.fetch = simple_fetch;
dp->version.any.instance = simple_instance;
```

13.3.5 Labels

Metrics and instances can be further described through the use of metadata labels, which are arbitrary name:value pairs associated with individual metrics and instances. There are several applications of this concept, but one of the most important is the ability to differentiate the components of a multi-dimensional instance name, such as the case of the **mem.zoneinfo.numa_hit** metric which has one value per memory zone, per NUMA node.

Consider *Example 2.11. Multi-dimensional Instance Domain Labels*:

Example 2.11. Multi-dimensional Instance Domain Labels

```
$ pminfo -l mem.zoneinfo.numa_hit

mem.zoneinfo.numa_hit
   inst [0 or "DMA::node0"] labels {"device_type":["numa_node","memory"],"indom_name":
↪"per zone per numa_node","numa_node":0,"zone":"DMA"}
   inst [1 or "Normal::node0"] labels {"device_type":["numa_node","memory"],"indom_
↪name":"per zone per numa_node","numa_node":0,"zone":"Normal"}
   inst [2 or "DMA::node1"] labels {"device_type":["numa_node","memory"],"indom_name":
↪"per zone per numa_node","numa_node":1,"zone":"DMA"}
   inst [3 or "Normal::node1"] labels {"device_type":["numa_node","memory"],"indom_
↪name":"per zone per numa_node","numa_node":1,"zone":"Normal"}
```

Note: The metric labels used here individually describe the memory zone and NUMA node associated with each instance.

The PMDA implementation is only partially responsible for choosing the label identifiers that differentiate components of metrics and instances within an instance domain. Label sets for a singleton metric or individual instance of a set-valued metric are formed from a label hierarchy, which includes global labels applied to all metrics and instances from one PMAPI context.

Labels are stored and communicated within PCP using JSONB format. This format is a restricted form of JSON suitable for indexing and other operations. In JSONB form, insignificant whitespace is discarded, and the order of label names is not preserved. Within the PMCS a lexicographically sorted key space is always maintained, however. Duplicate label names are not permitted. The label with highest precedence is the only one presented. If duplicate names are presented at the same hierarchy level, only one will be preserved (exactly which one wins is arbitrary, so do not rely on this).

Label Hierarchy

The set of labels associated with any singleton metric or instance is formed by merging the sets of labels at each level of a hierarchy. The lower levels of the hierarchy have highest precedence when merging overlapping (duplicate) label names:

- Global context labels (as reported by the **pmcd.labels** metric) are the lowest precedence. The PMDA implementor has no influence over labels at this level of the hierarchy, and these labels are typically supplied by **pmcd** from **/etc/pcp/labels** files.
- Domain labels, for all metrics and instances of a PMDA, are the next highest precedence.
- Instance Domain labels, associated with an InDom, are the next highest precedence.
- Metric cluster labels, associated with a PMID cluster, are the next highest precedence.
- Metric item labels, associated with an individual PMID, are the next highest precedence.
- Instance labels, associated with a metric instance identifier, have the highest precedence.

Data Structures

In any PMDA that supports labels at any level of the hierarchy, each individual label (one name:value pair) requires a **pmLabel** structure as shown in *Example 2.12. pmLabel Structure*:

Example 2.12. pmLabel Structure

```
typedef struct {
    uint     name : 16;      /* label name offset in JSONB string */
    uint     namelen : 8;    /* length of name excluding the null */
    uint     flags : 8;      /* information about this label */
    uint     value : 16;     /* offset of the label value */
    uint     valuelen : 16;  /* length of value in bytes */
} pmLabel;
```

The **flags** field is a bitfield identifying the hierarchy level and whether this name:value pair is intrinsic (optional) or extrinsic (part of the mandatory, identifying metadata for the metric or instance). All other fields are offsets and lengths in the JSONB string from an associated **pmLabelSet** structure.

Zero or more labels are specified via a label set, in a **pmLabelSet** structure as shown in *Example 2.13. pmLabelSet Structure*:

Example 2.13. pmLabelSet Structure

```
typedef struct {
    uint     inst;          /* PM_IN_NULL or the instance ID */
    int      nlabels;       /* count of labels or error code */
    char     *json;         /* JSONB formatted labels string */
    uint     jsonlen : 16;  /* JSON string length byte count */
    uint     padding : 16;  /* zero, reserved for future use */
    pmLabel  *labels;       /* indexing into the JSON string */
} pmLabelSet;
```

This provides information about the set of labels associated with an entity (context, domain, indom, metric cluster, item or instance). The entity will be from any one level of the label hierarchy. If at the lowest hierarchy level (which happens to be highest precedence - instances) then the **inst** field will contain an actual instance identifier instead of PM_IN_NULL.

For information about how a label can be associated with each level of the hierarchy, see the **pmdaLabel(3)** man page.

The simple PMDA, shown in *Example 2.14. Simple PMDA*, associates labels at the domain, indom and instance levels of the hierarchy.

Example 2.14. Simple PMDA

```
static int
simple_label(int ident, int type, pmLabelSet **lpp, pmdaExt *pmda)
{
    int         serial;

    switch (type) {
    case PM_LABEL_DOMAIN:
        pmdaAddLabels(lpp, "{"role":"testing"}");
        break;
    case PM_LABEL_INDOM:
        serial = pmInDom_serial((pmInDom)ident);
        if (serial == COLOR_INDOM) {
            pmdaAddLabels(lpp, "{"indom_name":"color"}");
            pmdaAddLabels(lpp, "{"model":"RGB"}");
```

(continues on next page)

(continued from previous page)

```
        }
        if (serial == NOW_INDOM) {
            pmdaAddLabels(lpp, "{"indom_name":"time"}");
            pmdaAddLabels(lpp, "{"unitsystem":"SI"}");
        }
        break;
    case PM_LABEL_CLUSTER:
    case PM_LABEL_ITEM:
        /* no labels to add for these types, fall through */
    default:
        break;
    }
    return pmdaLabel(ident, type, lpp, pmda);
}

static int
simple_labelCallBack(pmInDom indom, unsigned int inst, pmLabelSet **lp)
{
    struct timeslice *tsp;

    if (pmInDom_serial(indom) != NOW_INDOM)
        return 0;
    if (pmdaCacheLookup(indom, inst, NULL, (void *)&tsp) != PMDA_CACHE_ACTIVE)
        return 0;
    /* SI units label, value: sec (seconds), min (minutes), hour (hours) */
    return pmdaAddLabels(lp, "{"units":"%s"}", tsp-<tm_name);
}
```

The **simple_labelCallBack** function is called indirectly via **pmdaLabel** for each instance of the **NOW_INDOM**. PMDA initialization ensures these functions are registered with the global PMDA interface structure for use when handling label requests, by the following assignments in the **simple_init** function:

```
dp->version.seven.label = simple_label;
pmdaSetLabelCallBack(dp, simple_labelCallBack);
```

13.4 Other Issues

Other issues include extracting the information, latency and threads of control, Name Space, PMDA help text, and management of evolution within a PMDA.

13.4.1 Extracting the Information

A suggested approach to writing a PMDA is to write a standalone program to extract the values from the target domain and then incorporate this program into the PMDA framework. This approach avoids concurrent debugging of two distinct problems:

1. Extraction of the data
2. Communication with PMCD

These are some possible ways of exporting the data from the target domain:

- Accumulate the performance data in a public shared memory segment.
- Write the performance data to the end of a log file.

- Periodically rewrite a file with the most recent values for the performance data.
- Implement a protocol that allows a third party to connect to the target application, send a request, and receive new performance data.
- If the data is in the operating system kernel, provide a kernel interface (preferred) to export the performance data.

Most of these approaches require some further data processing by the PMDA.

13.4.2 Latency and Threads of Control

The PCP protocols expect PMDAs to return the current values for performance metrics when requested, and with short delay (low latency). For some target domains, access to the underlying instrumentation may be costly or involve unpredictable delays (for example, if the real performance data is stored on some remote host or network device). In these cases, it may be necessary to separate probing for new performance data from servicing PMCD requests.

An architecture that has been used successfully for several PMDAs is to create one or more child processes to obtain information while the main process communicates with PMCD.

At the simplest deployment of this arrangement, the two processes may execute without synchronization. Threads have also been used as a more portable multithreading mechanism; see the **pthreads(7)** man page.

By contrast, a complex deployment would be one in which the refreshing of the metric values must be atomic, and this may require double buffering of the data structures. It also requires coordination between parent and child processes.

Warning: Since certain data structures used by the PMDA library are not thread-aware, only one PMDA thread of control should call PMDA library functions - this would typically be the thread servicing requests from PMCD.

One caveat about this style of caching PMDA–in this (special) case it is better if the PMDA converts counts to rates based upon consecutive periodic sampling from the underlying instrumentation. By exporting precomputed rate metrics with instantaneous semantics, the PMDA prevents the PCP monitor tools from computing their own rates upon consecutive PMCD fetches (which are likely to return identical values from a caching PMDA). The finer points of metric semantics are discussed in Section 2.3.3.2, "*Semantics*"

13.4.3 Name Space

The PMNS file defines the name space of the PMDA. It is a simple text file that is used during installation to expand the Name Space of the PMCD process. The format of this file is described by the **pmns(5)** man page and its hierarchical nature, syntax, and helper tools are further described in the *Performance Co-Pilot User's and Administrator's Guide*.

Client processes will not be able to access the PMDA metrics if the PMNS file is not installed as part of the PMDA installation procedure on the collector host. The installed list of metric names and their corresponding PMIDs can be found in `${PCP_VAR_DIR}/pmns/root`.

Example 2.15. pmns File for the Simple PMDA shows the simple PMDA, which has five metrics:

Three metrics immediately under the **simple** node

Two metrics under another non-terminal node called **simple.time**

Example 2.15. pmns File for the Simple PMDA

```
simple {
    numfetch    SIMPLE:0:0
    color       SIMPLE:0:1
```

(continues on next page)

(continued from previous page)

```
    time
    now         SIMPLE:2:4
}
simple.time {
    user        SIMPLE:1:2
    sys         SIMPLE:1:3
}
```

Metrics that have different clusters do not have to be specified in different subtrees of the PMNS. *Example 2.16. Alternate pmns File for the Simple PMDA* shows an alternative PMNS for the simple PMDA:

Example 2.16. Alternate pmns File for the Simple PMDA

```
simple {
    numfetch    SIMPLE:0:0
    color       SIMPLE:0:1
    usertime    SIMPLE:1:2
    systime     SIMPLE:1:3
}
```

In this example, the **SIMPLE** macro is replaced by the domain number listed in `${PCP_VAR_DIR}/pmns/stdpmid` for the corresponding PMDA during installation (for the simple PMDA, this would normally be the value 253).

If the PMDA implementer so chooses, all or a subset of the metric names and identifiers can be specified programatically. In this situation, a special asterisk syntax is used to denote those subtrees which are to be handles this way. *Example 2.17. Dynamic metrics pmns File for the Simple PMDA* shows this dynamic namespace syntax, for all metrics in the simple PMDA:

Example 2.17. Dynamic metrics pmns File for the Simple PMDA

```
simple          SIMPLE:*:*
```

In this example, like the one before, the **SIMPLE** macro is replaced by the domain number, and all (simple.*) metric namespace operations must be handled by the PMDA. This is in contrast to the static metric name model earlier, where the host-wide PMNS file is updated and used by PMCD, acting on behalf of the agent.

13.4.4 PMDA Help Text

For each metric defined within a PMDA, the PMDA developer is strongly encouraged to provide both terse and extended help text to describe the metric, and perhaps provide hints about the expected value ranges.

The help text is used to describe each metric in the visualization tools and **pminfo** with the **-T** option. The help text, such as the help text for the simple PMDA in *Example 2.18. Help Text for the Simple PMDA*, is specified in a specially formatted file, normally called **help**. This file is converted to the expected run-time format using the **newhelp** command; see the **newhelp(1)** man page. Converted help text files are usually placed in the PMDA's directory below `${PCP_PMDAS_DIR}` as part of the PMDA installation procedure.

Example 2.18. Help Text for the Simple PMDA

The two instance domains and five metrics have a short and a verbose description. Each entry begins with a line that starts with the character "@" and is followed by either the metric name (**simple.numfetch**) or a symbolic reference to the instance domain number (**SIMPLE.1**), followed by the short description. The verbose description is on the following lines, terminated by the next line starting with "@" or end of file:

```
@ SIMPLE.0 Instance domain "colour" for simple PMDA
Universally 3 instances, "red" (0), "green" (1) and "blue" (3).

@ SIMPLE.1 Dynamic instance domain "time" for simple PMDA
An instance domain is computed on-the-fly for exporting current time
information. Refer to the help text for simple.now for more details.

@ simple.numfetch Number of pmFetch operations.
The cumulative number of pmFetch operations directed to "simple" PMDA.

This counter may be modified with pmstore(1).

@ simple.color Metrics which increment with each fetch
This metric has 3 instances, designated "red", "green" and "blue".

The value of the metric is monotonic increasing in the range 0 to
255, then back to 0.  The different instances have different starting
values, namely 0 (red), 100 (green) and 200 (blue).

The metric values my be altered using pmstore(1).

@ simple.time.user Time agent has spent executing user code
The time in seconds that the CPU has spent executing agent user code.

@ simple.time.sys Time agent has spent executing system code
The time in seconds that the CPU has spent executing agent system code.

@ simple.now Time of day with a configurable instance domain
The value reflects the current time of day through a dynamically
reconfigurable instance domain.  On each metric value fetch request,
the agent checks to see whether the configuration file in
${PCP_PMDAS_DIR}/simple/simple.conf has been modified - if it has then
the file is re-parsed and the instance domain for this metric is again
constructed according to its contents.

This configuration file contains a single line of comma-separated time
tokens from this set:
  "sec"  (seconds after the minute),
  "min"  (minutes after the hour),
  "hour" (hour since midnight).

An example configuration file could be:  sec,min,hour
and in this case the simple.now metric would export values for the
three instances "sec", "min" and "hour" corresponding respectively to
the components seconds, minutes and hours of the current time of day.

The instance domain reflects each token present in the file, and the
values reflect the time at which the PMDA processes the fetch.
```

13.4.5 Management of Evolution within a PMDA

Evolution of a PMDA, or more particularly the underlying instrumentation to which it provides access, over time naturally results in the appearance of new metrics and the disappearance of old metrics. This creates potential problems for PMAPI clients and PCP tools that may be required to interact with both new and former versions of the PMDA.

The following guidelines are intended to help reduce the complexity of implementing a PMDA in the face of evolutionary change, while maintaining predictability and semantic coherence for tools using the PMAPI, and for end users of those tools.

- Try to support as full a range of metrics as possible in every version of the PMDA. In this context, *support* means responding sensibly to requests, even if the underlying instrumentation is not available.
- If a metric is not supported in a given version of the underlying instrumentation, the PMDA should respond to **pmLookupDesc** requests with a **pmDesc** structure whose **type** field has the special value **PM_TYPE_NOSUPPORT**. Values of fields other than **pmid** and **type** are immaterial, but *Example 2.19. Setting Values* is typically benign:

Example 2.19. Setting Values

```
pmDesc dummy = {
     .pmid  = PMDA_PMID(3,0),           /* pmid, fill this in */
     .type  = PM_TYPE_NOSUPPORT,        /* this is the important part */
     .indom = PM_INDOM_NULL,            /* singular,causes no problems */
     .sem   = 0,                        /* no semantics */
     .units = PMDA_PMUNITS(0,0,0,0,0,0) /* no units */
};
```

- If a metric lacks support in a particular version of the underlying instrumentation, the PMDA should respond to **pmFetch** requests with a **pmResult** in which no values are returned for the unsupported metric. This is marginally friendlier than the other semantically acceptable option of returning an illegal PMID error or **PM_ERR_PMID**.
- Help text should be updated with annotations to describe different versions of the underlying product, or product configuration options, for which a specific metric is available. This is so **pmLookupText** can always respond correctly.
- The **pmStore** operation should fail with return status of **PM_ERR_PERMISSION** if a user or application tries to amend the value of an unsupported metric.
- The value extraction, conversion, and printing functions (**pmExtractValue, pmConvScale, pmAtomStr, pmTypeStr**, and **pmPrintValue**) return the **PM_ERR_CONV** error or an appropriate diagnostic string, if an attempt is made to operate on a value for which **type** is **PM_TYPE_NOSUPPORT**.
- If performance tools take note of the **type** field in the **pmDesc** structure, they should not manipulate values for unsupported metrics. Even if tools ignore **type** in the metric's description, following these development guidelines ensures that no misleading value is ever returned; so there is no reason to call the extraction, conversion, and printing functions.

13.5 PMDA Interface

This section describes an interface for the request handling callbacks in a PMDA. This interface is used by PMCD for communicating with DSO PMDAs and is also used by daemon PMDAs with **pmdaMain**.

13.5.1 Overview

Both daemon and DSO PMDAs must handle multiple request types from PMCD. A daemon PMDA communicates with PMCD using the PDU protocol, while a DSO PMDA defines callbacks for each request type. To avoid duplicating this PDU processing (in the case of a PMDA that can be installed either as a daemon or as a DSO), and to allow a consistent framework, **pmdaMain** can be used by a daemon PMDA as a wrapper to handle the communication protocol using the same callbacks as a DSO PMDA. This allows a PMDA to be built as both a daemon and a DSO, and then to be installed as either.

To further simplify matters, default callbacks are declared in **<pcp/pmda.h>**:

pmdaFetch
pmdaProfile
pmdaInstance
pmdaDesc
pmdaText
pmdaStore
pmdaPMID
pmdaName
pmdaChildren
pmdaAttribute
pmdaLabel

Each callback takes a **pmdaExt** structure as its last argument. This structure contains all the information that is required by the default callbacks in most cases. The one exception is **pmdaFetch**, which needs an additional callback to instantiate the current value for each supported combination of a performance metric and an instance.

Therefore, for most PMDAs all the communication with PMCD is automatically handled by functions in **libpcp.so** and **libpcp_pmda.so**.

Trivial PMDA

The trivial PMDA uses all of the default callbacks as shown in *Example 2.20. Request Handling Callbacks in the Trivial PMDA*. The additional callback for **pmdaFetch** is defined as **trivial_fetchCallBack**:

Example 2.20. Request Handling Callbacks in the Trivial PMDA

```
static int
trivial_fetchCallBack(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
   __pmID_int      *idp = (__pmID_int *)&(mdesc->m_desc.pmid);

   if (idp->cluster != 0 || idp->item != 0)
       return PM_ERR_PMID;
   if (inst != PM_IN_NULL)
       return PM_ERR_INST;
   atom->l = time(NULL);
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

This function checks that the PMID and instance are valid, and then places the metric value for the current time into the **pmAtomValue** structure.

The callback is set up by a call to **pmdaSetFetchCallBack** in **trivial_init**. As a rule of thumb, the API routines with named ending with **CallBack** are helpers for the higher PDU handling routines like **pmdaFetch**. The latter are set directly using the PMDA Interface Structures, as described in Section 2.5.2, "*PMDA Structures*".

Simple PMDA

The simple PMDA callback for **pmdaFetch** is more complicated because it supports more metrics, some metrics are instantiated with each fetch, and one instance domain is dynamic. The default **pmdaFetch** callback, shown in *Example 2.21. Request Handling Callbacks in the Simple PMDA*, is replaced by **simple_fetch** in **simple_init**, which increments the number of fetches and updates the instance domain for **INDOM_NOW** before calling **pmdaFetch**:

Example 2.21. Request Handling Callbacks in the Simple PMDA

```
static int
simple_fetch(int numpmid, pmID pmidlist[], pmResult **resp, pmdaExt *pmda)
{
    numfetch++;
    simple_timenow_check();
    simple_timenow_refresh();
    return pmdaFetch(numpmid, pmidlist, resp, pmda);
}
```

The callback for **pmdaFetch** is defined as **simple_fetchCallBack**. The PMID is extracted from the **pmdaMetric** structure, and if valid, the appropriate field in the **pmAtomValue** structure is set. The available types and associated fields are described further in Section 3.4, "*Performance Metric Descriptions*" and *Example 3.18. pmAtomValue Structure*.

Note: Note that PMID validity checking need only check the cluster and item numbers, the domain number is guaranteed to be valid and the PMDA should make no assumptions about the actual domain number being used at this point.

The **simple.numfetch** metric has no instance domain and is easily handled first as shown in *Example 2.22. simple.numfetch Metric*:

Example 2.22. simple.numfetch Metric

```
static int
simple_fetchCallBack(pmdaMetric *mdesc, unsigned int inst, pmAtomValue *atom)
{
   int             i;
   static int      oldfetch;
   static double   usr, sys;
   __pmID_int      *idp = (__pmID_int *)&(mdesc->m_desc.pmid);

   if (inst != PM_IN_NULL &&
       !(idp->cluster == 0 && idp->item == 1) &&
       !(idp->cluster == 2 && idp->item == 4))
       return PM_ERR_INST;
```

(continues on next page)

(continued from previous page)

```
   if (idp->cluster == 0) {
       if (idp->item == 0) {                    /* simple.numfetch */
           atom->l = numfetch;
       }
```

In *Example 2.23. simple.color Metric*, the **inst** parameter is used to specify which instance is required for the **simple.color** metric:

Example 2.23. simple.color Metric

```
else if (idp->item == 1) {              /* simple.color */
      switch (inst) {
      case 0:                             /* red */
          red = (red + 1) % 256;
          atom->l = red;
          break;
      case 1:                             /* green */
          green = (green + 1) % 256;
          atom->l = green;
          break;
      case 2:                             /* blue */
          blue = (blue + 1) % 256;
          atom->l = blue;
          break;
      default:
          return PM_ERR_INST;
      }
 }
 else
     return PM_ERR_PMID;
```

In *Example 2.24. simple.time Metric*, the **simple.time** metric is in a second cluster and has a simple optimization to reduce the overhead of calling **times** twice on the same fetch and return consistent values from a single call to **times** when both metrics **simple.time.user** and **simple.time.sys** are requested in a single **pmFetch**. The previous fetch count is used to determine if the **usr** and **sys** values should be updated:

Example 2.24. simple.time Metric

```
else if (idp->cluster == 1) {               /* simple.time */
    if (oldfetch < numfetch) {
        __pmProcessRunTimes(&usr, &sys);
        oldfetch = numfetch;
    }
    if (idp->item == 2)                     /* simple.time.user */
        atom->d = usr;
    else if (idp->item == 3)                /* simple.time.sys */
        atom->d = sys;
    else
        return PM_ERR_PMID;
 }
```

In *Example 2.25. simple.now Metric*, the **simple.now** metric is in a third cluster and uses **inst** again to select a specific instance from the **INDOM_NOW** instance domain. The values associated with instances in this instance domain are managed using the **pmdaCache(3)** helper routines, which provide efficient interfaces for managing more complex instance domains:

Example 2.25. simple.now Metric

```
else if (idp->cluster == 2) {
    if (idp->item == 4) {                /* simple.now */
        struct timeslice *tsp;
        sts = pmdaCacheLookup(*now_indom, inst, NULL, (void *)&tsp);
        if (sts != PMDA_CACHE_ACTIVE) {
            if (sts < 0)
                pmNotifyErr(LOG_ERR, "pmdaCacheLookup failed: inst=%d: %s",
                            inst, pmErrStr(sts));
            return PM_ERR_INST;
        }
        atom->l = tsp->tm_field;
    }
    else
        return PM_ERR_PMID;
}
```

simple_store in the Simple PMDA

The simple PMDA permits some of the metrics it supports to be modified by **pmStore** as shown in *Example 2.26. simple_store in the Simple PMDA*. For additional information, see the **pmstore(1)** and **pmStore(3)** man pages.

Example 2.26. simple_store in the Simple PMDA

The **pmdaStore** callback (which returns **PM_ERR_PERMISSION** to indicate no metrics can be altered) is replaced by **simple_store** in **simple_init**. This replacement function must take the same arguments so that it can be assigned to the function pointer in the **pmdaInterface** structure.

The function traverses the **pmResult** and checks the cluster and unit of each PMID to ensure that it corresponds to a metric that can be changed. Checks are made on the values to ensure they are within range before being assigned to variables in the PMDA that hold the current values for exported metrics:

```
static int
simple_store(pmResult *result, pmdaExt *pmda)
{
    int         i, j, val, sts = 0;
    pmAtomValue av;
    pmValueSet  *vsp = NULL;
    __pmID_int  *pmidp = NULL;

    /* a store request may affect multiple metrics at once */
    for (i = 0; i < result->numpmid; i++) {
        vsp = result->vset[i];
        pmidp = (__pmID_int *)&vsp->pmid;
        if (pmidp->cluster == 0) {  /* storable metrics are cluster 0 */
            switch (pmidp->item) {
            case 0:                             /* simple.numfetch */
                val = vsp->vlist[0].value.lval;
                if (val < 0) {
                    sts = PM_ERR_SIGN;
                    val = 0;
                }
                numfetch = val;
                break;
            case 1:                              /* simple.color */
                /* a store request may affect multiple instances at once */
                for (j = 0; j < vsp->numval && sts == 0; j++) {
```

(continues on next page)

(continued from previous page)

```
                    val = vsp->vlist[j].value.lval;
                    if (val < 0) {
                        sts = PM_ERR_SIGN;
                        val = 0;
                    } if (val > 255) {
                        sts = PM_ERR_CONV;
                        val = 255;
                    }
```

The **simple.color** metric has an instance domain that must be searched because any or all instances may be specified. Any instances that are not supported in this instance domain should cause an error value of **PM_ERR_INST** to be returned as shown in *Example 2.27. simple.color and PM_ERR_INST Errors*:

Example 2.27. simple.color and PM_ERR_INST Errors

```
switch (vsp->vlist[j].inst) {
case 0:                         /* red */
    red = val;
    break;
case 1:                         /* green */
    green = val;
    break;
case 2:                         /* blue */
    blue = val;
    break;
default:
    sts = PM_ERR_INST;
}
```

Any other PMIDs in cluster 0 that are not supported by the simple PMDA should result in an error value of **PM_ERR_PMID** as shown in *Example 2.28. PM_ERR_PMID Errors*:

Example 2.28. PM_ERR_PMID Errors

```
        default:
            sts = PM_ERR_PMID;
            break;
    }
}
```

Any metrics that cannot be altered should generate an error value of **PM_ERR_PERMISSION**, and metrics not supported by the PMDA should result in an error value of **PM_ERR_PMID** as shown in *Example 2.29. PM_ERR_PERMISSION and PM_ERR_PMID Errors*:

Example 2.29. PM_ERR_PERMISSION and PM_ERR_PMID Errors

```
        else if ((pmidp->cluster == 1 &&
                  (pmidp->item == 2 || pmidp->item == 3)) ||
                  (pmidp->cluster == 2 && pmidp->item == 4)) {
            sts = PM_ERR_PERMISSION;
            break;
        }
        else {
            sts = PM_ERR_PMID;
            break;
        }
    }
```

(continues on next page)

(continued from previous page)

```
    return sts;
}
```

The structure **pmdaExt** *pmda* argument is not used by the **simple_store** function above.

Note: When using storable metrics, it is important to consider the implications. It is possible **pmlogger** is actively sampling the metric being modified, for example, which may cause unexpected results to be persisted in an archive. Consider also the use of client credentials, available via the **attribute** callback of the **pmdaInterface** structure, to appropriately limit access to any modifications that might be made via your storable metrics.

Return Codes for pmdaFetch Callbacks

In **PMDA_INTERFACE_1** and **PMDA_INTERFACE_2**, the return codes for the **pmdaFetch** callback function are defined:

Value	Meaning
< 0	Error code (for example, **PM_ERR_PMID, PM_ERR_INST** or **PM_ERR_AGAIN**)
0	Success

In **PMDA_INTERFACE_3** and all later versions, the return codes for the **pmdaFetch** callback function are defined:

Value	Meaning
< 0	Error code (for example, **PM_ERR_PMID, PM_ERR_INST**)
0	Metric value not currently available
> 0	Success

13.5.2 PMDA Structures

PMDA structures used with the **pcp_pmda** library are defined in **<pcp/pmda.h>**. *Example 2.30. pmdaInterface Structure Header* and *Example 2.32. pmdaExt Structure* describe the **pmdaInterface** and **pmdaExt** structures.

Example 2.30. pmdaInterface Structure Header

The callbacks must be specified in a **pmdaInterface** structure:

```
typedef struct {
    int domain;     /* set/return performance metrics domain id here */
    struct {
        unsigned int pmda_interface : 8;  /* PMDA DSO version */
        unsigned int pmapi_version : 8;   /* PMAPI version */
        unsigned int flags : 16;          /* optional feature flags */
    } comm;             /* set/return communication and version info */
    int status;         /* return initialization status here */
    union {
        ...
```

This structure is passed by PMCD to a DSO PMDA as an argument to the initialization function. This structure supports multiple (binary-compatible) versions–the second and subsequent versions have support for the **pmdaExt** structure. Protocol version one is for backwards compatibility only, and should not be used in any new PMDA.

To date there have been six revisions of the interface structure:

1. Version two added the **pmdaExt** structure, as mentioned above.
2. Version three changed the fetch callback return code semantics, as mentioned in Section 2.5.1.4, "*Return Codes for pmdaFetch Callbacks*".
3. Version four added support for dynamic metric names, where the PMDA is able to create and remove metric names on-the-fly in response to changes in the performance domain (**pmdaPMID, pmdaName, pmdaChildren** interfaces)
4. Version five added support for per-client contexts, where the PMDA is able to track arrival and disconnection of PMAPI client tools via PMCD (**pmdaGetContext** helper routine). At the same time, support for **PM_TYPE_EVENT** metrics was implemented, which relies on the per-client context concepts (**pmdaEvent*** helper routines).
5. Version six added support for authenticated client contexts, where the PMDA is informed of user credentials and other PMCD attributes of the connection between individual PMAPI clients and PMCD (**pmdaAttribute** interface)
6. Version seven added support for metadata labels, where the PMDA is able to associate name:value pairs in a hierarchy such that additional metadata, above and beyond the metric descriptors, is associated with metrics and instances (**pmdaLabel** interface)

Example 2.31. pmdaInterface Structure, Latest Version

```
   ...
    union {
        ...
        /*
         * PMDA_INTERFACE7
         */
        struct {
            pmdaExt *ext;
            int     (*profile)(pmdaInProfile *, pmdaExt *);
            int     (*fetch)(int, pmID *, pmResult **, pmdaExt *);
            int     (*desc)(pmID, pmDesc *, pmdaExt *);
            int     (*instance)(pmInDom, int, char *, pmdaInResult **, pmdaExt *);
            int     (*text)(int, int, char **, pmdaExt *);
            int     (*store)(pmResult *, pmdaExt *);
            int     (*pmid)(const char *, pmID *, pmdaExt *);
            int     (*name)(pmID, char ***, pmdaExt *);
            int     (*children)(const char *, int, char ***, int **, pmdaExt *);
            int     (*attribute)(int, int, const char *, int, pmdaExt *);
            int     (*label)(int, int, pmLabelSet **, pmdaExt *);
        } seven;
    } version;
} pmdaInterface;
```

Note: Each new interface version is always defined as a superset of those that preceded it, only adds fields at the end of the new structure in the union, and is always binary backwards-compatible. **And thus it shall remain**. For brevity, we have shown only the latest interface version (seven) above, but all prior versions still exist, build, and function. In other words, PMDAs built against earlier versions of this header structure (and PMDA library) function correctly with the latest version of the PMDA library.

Example 2.32. pmdaExt Structure

Additional PMDA information must be specified in a **pmdaExt** structure:

```
typedef struct {
    unsigned int e_flags;       /* PMDA_EXT_FLAG_* bit field */
    void        *e_ext;         /* used internally within libpcp_pmda */
    char        *e_sockname;    /* socket name to pmcd */
    char        *e_name;        /* name of this pmda */
    char        *e_logfile;     /* path to log file */
    char        *e_helptext;    /* path to help text */
    int         e_status;       /* =0 is OK */
    int         e_infd;         /* input file descriptor from pmcd */
    int         e_outfd;        /* output file descriptor to pmcd */
    int         e_port;         /* port to pmcd */
    int         e_singular;     /* =0 for singular values */
    int         e_ordinal;      /* >=0 for non-singular values */
    int         e_direct;       /* =1 if pmid map to meta table */
    int         e_domain;       /* metrics domain */
    int         e_nmetrics;     /* number of metrics */
    int         e_nindoms;      /* number of instance domains */
    int         e_help;         /* help text comes via this handle */
    pmProfile   *e_prof;        /* last received profile */
    pmdaIoType  e_io;           /* connection type to pmcd */
    pmdaIndom   *e_indoms;      /* instance domain table */
    pmdaIndom   *e_idp;         /* instance domain expansion */
    pmdaMetric  *e_metrics;     /* metric description table */
    pmdaResultCallBack e_resultCallBack; /* to clean up pmResult after fetch */
    pmdaFetchCallBack  e_fetchCallBack;  /* to assign metric values in fetch */
    pmdaCheckCallBack  e_checkCallBack;  /* callback on receipt of a PDU */
    pmdaDoneCallBack   e_doneCallBack;   /* callback after PDU is processed */
    /* added for PMDA_INTERFACE_5 */
    int         e_context;      /* client context id from pmcd */
    pmdaEndContextCallBack e_endCallBack;  /* callback after client context closed */
    /* added for PMDA_INTERFACE_7 */
    pmdaLabelCallBack  e_labelCallBack;  /* callback to lookup metric instance labels␣
↪*/
} pmdaExt;
```

The **pmdaExt** structure contains filenames, pointers to tables, and some variables shared by several functions in the **pcp_pmda** library. All fields of the **pmdaInterface** and **pmdaExt** structures can be correctly set by PMDA initialization functions; see the **pmdaDaemon(3)**, **pmdaDSO(3)**, **pmdaGetOptions(3)**, **pmdaInit(3)**, and **pmdaConnect(3)** man pages for a full description of how various fields in these structures may be set or used by **pcp_pmda** library functions.

13.6 Initializing a PMDA

Several functions are provided to simplify the initialization of a PMDA. These functions, if used, must be called in a strict order so that the PMDA can operate correctly.

13.6.1 Overview

The initialization process for a PMDA involves opening help text files, assigning callback function pointers, adjusting the metric and instance identifiers to the correct domains, and much more. The initialization of a daemon PMDA also differs significantly from a DSO PMDA, since the **pmdaInterface** structure is initialized by **main** or the PMCD process, respectively.

13.6.2 Common Initialization

As described in Section 2.2.2, "*DSO PMDA*", an initialization function is provided by a DSO PMDA and called by PMCD. Using the standard PMDA wrappers, the same function can also be used as part of the daemon PMDA initialization. This PMDA initialization function performs the following tasks:

- Assigning callback functions to the function pointer interface of **pmdaInterface**
- Assigning pointers to the metric and instance tables from **pmdaExt**
- Opening the help text files
- Assigning the domain number to the instance domains
- Correlating metrics with their instance domains

If the PMDA uses the common data structures defined for the **pcp_pmda** library, most of these requirements can be handled by the default **pmdaInit** function; see the **pmdaInit(3)** man page.

Because the initialization function is the only initialization opportunity for a DSO PMDA, the common initialization function should also perform any DSO-specific functions that are required. A default implementation of this functionality is provided by the **pmdaDSO** function; see the **pmdaDSO(3)** man page.

Trivial PMDA

Example 2.33. Initialization in the Trivial PMDA shows the trivial PMDA, which has no instances (that is, all metrics have singular values) and a single callback. This callback is for the **pmdaFetch** function called **trivial_fetchCallBack**; see the **pmdaFetch(3)** man page:

Example 2.33. Initialization in the Trivial PMDA

```
static char     *username;
static int      isDSO = 1;              /* ==0 if I am a daemon */

void trivial_init(pmdaInterface *dp)
{
    if (isDSO)
        pmdaDSO(dp, PMDA_INTERFACE_2, "trivial DSO",
                "${PCP_PMDAS_DIR}/trivial/help");
    else
        pmSetProcessIdentity(username);

    if (dp->status != 0)
        return;

    pmdaSetFetchCallBack(dp, trivial_fetchCallBack);
    pmdaInit(dp, NULL, 0,
             metrictab, sizeof(metrictab)/sizeof(metrictab[0]));
}
```

The trivial PMDA can execute as either a DSO or daemon PMDA. A default installation installs it as a daemon, however, and the **main** routine clears *isDSO* and sets *username* accordingly.

The **trivial_init** routine provides the opportunity to do any extra DSO or daemon setup before calling the library **pmdaInit**. In the example, the help text is setup for DSO mode and the daemon is switched to run as an unprivileged user (default is **root**, but it is generally good form for PMDAs to run with the least privileges possible). If **dp->status** is non-zero after the **pmdaDSO** call, the PMDA will be removed by PMCD and cannot safely continue to use the **pmdaInterface** structure.

Simple PMDA

In *Example 2.34. Initialization in the Simple PMDA*, the simple PMDA uses its own callbacks to handle **PDU_FETCH** and **PDU_RESULT** request PDUs (for **pmFetch** and **pmStore** operations respectively), as well as providing **pmdaFetch** with the callback **simple_fetchCallBack**.

Example 2.34. Initialization in the Simple PMDA

```
static int      isDSO = 1;              /* =0 I am a daemon */
static char     *username;

void simple_init(pmdaInterface *dp)
{
    if (isDSO)
        pmdaDSO(dp, PMDA_INTERFACE_7, "simple DSO",
                "${PCP_PMDAS_DIR}/simple/help");
    else
        pmSetProcessIdentity(username);

    if (dp->status != 0)
        return;

    dp->version.any.fetch = simple_fetch;
    dp->version.any.store = simple_store;
    dp->version.any.instance = simple_instance;
    dp->version.seven.label = simple_label;
    pmdaSetFetchCallBack(dp, simple_fetchCallBack);
    pmdaSetLabelCallBack(dp, simple_labelCallBack);
    pmdaInit(dp, indomtab, sizeof(indomtab)/sizeof(indomtab[0]),
             metrictab, sizeof(metrictab)/sizeof(metrictab[0]));
}
```

Once again, the simple PMDA may be installed either as a daemon PMDA or a DSO PMDA. The static variable *isDSO* indicates whether the PMDA is running as a DSO or as a daemon. A daemon PMDA always changes the value of this variable to 0 in *main*, for PMDAs that can operate in both modes.

Remember also, as described earlier, **simple_fetch** is dealing with a single request for (possibly many) values for metrics from the PMDA, and **simple_fetchCallBack** is its little helper, dealing with just one metric and one instance (optionally, if the metric happens to have an instance domain) within that larger request.

13.6.3 Daemon Initialization

In addition to the initialization function that can be shared by a DSO and a daemon PMDA, a daemon PMDA must also meet the following requirements:

- Create the **pmdaInterface** structure that is passed to the initialization function
- Parse any command-line arguments
- Open a log file (a DSO PMDA uses PMCD's log file)
- Set up the IPC connection between the PMDA and the PMCD process
- Handle incoming PDUs

All these requirements can be handled by default initialization functions in the **pcp_pmda** library; see the **pmdaDaemon(3), pmdaGetOptions(3), pmdaOpenLog(3), pmdaConnect(3)**, and **pmdaMain(3)** man pages.

Note: Optionally, a daemon PMDA may wish to reduce or change its privilege level, as seen in *Example 2.33. Initialization in the Trivial PMDA* and *Example 2.34. Initialization in the Simple PMDA*. Some performance domains **require** the extraction process to run as a specific user in order to access the instrumentation. Many domains require the default **root** level of access for a daemon PMDA.

The simple PMDA specifies the command-line arguments it accepts using **pmdaGetOptions**, as shown in *Example 2.35. main in the Simple PMDA*. For additional information, see the **pmdaGetOptions(3)** man page.

Example 2.35. main in the Simple PMDA

```
static pmLongOptions longopts[] = {
    PMDA_OPTIONS_HEADER("Options"),
    PMOPT_DEBUG,
    PMDAOPT_DOMAIN,
    PMDAOPT_LOGFILE,
    PMDAOPT_USERNAME,
    PMOPT_HELP,
    PMDA_OPTIONS_TEXT("\nExactly one of the following options may appear:"),
    PMDAOPT_INET,
    PMDAOPT_PIPE,
    PMDAOPT_UNIX,
    PMDAOPT_IPV6,
    PMDA_OPTIONS_END
};
static pmdaOptions opts = {
    .short_options = "D:d:i:l:pu:U:6:?",
    .long_options = longopts,
};

int
main(int argc, char **argv)
{
    pmdaInterface       dispatch;

    isDSO = 0;
    pmSetProgname(argv[0]);
    pmGetUsername(&username);
    pmdaDaemon(&dispatch, PMDA_INTERFACE_7, pmGetProgname(), SIMPLE,
               "simple.log", "${PCP_PMDAS_DIR}/simple/help");
```

(continues on next page)

(continued from previous page)

```
    pmdaGetOptions(argc, argv, &opts, &dispatch);
    if (opts.errors) {
        pmdaUsageMessage(&opts);
        exit(1);
    }
    if (opts.username)
        username = opts.username;

    pmdaOpenLog(&dispatch);
    simple_init(&dispatch);
    simple_timenow_check();
    pmdaConnect(&dispatch);
    pmdaMain(&dispatch);

    exit(0);
}
```

The conditions under which **pmdaMain** will return are either unexpected error conditions (often from failed initialisation, which would already have been logged), or when PMCD closes the connection to the PMDA. In all cases the correct action to take is simply to exit cleanly, possibly after any final cleanup the PMDA may need to perform.

13.7 Testing and Debugging a PMDA

Ensuring the correct operation of a PMDA can be difficult, because the responsibility of providing metrics to the requesting PMCD process and simultaneously retrieving values from the target domain requires nearly real-time communication with two modules beyond the PMDA's control. Some tools are available to assist in this important task.

13.7.1 Overview

Thoroughly testing a PMDA with PMCD is difficult, although testing a daemon PMDA is marginally simpler than testing a DSO PMDA. If a DSO PMDA exits, PMCD also exits because they share a single address space and control thread.

The difficulty in using PMCD to test a daemon PMDA results from PMCD requiring timely replies from the PMDA in response to request PDUs. Although a timeout period can be set in `${PCP_PMCDOPTIONS_PATH}`, attaching a debugger (such as **gdb**) to the PMDA process might cause an already running PMCD to close its connection with the PMDA. If timeouts are disabled, PMCD could wait forever to connect with the PMDA.

If you suspect a PMDA has been terminated due to a timeout failure, check the PMCD log file, usually `${PCP_LOG_DIR}/pmcd/pmcd.log`.

A more robust way of testing a PMDA is to use the **dbpmda** tool, which is similar to PMCD except that **dbpmda** provides complete control over the PDUs that are sent to the PMDA, and there are no time limits–it is essentially an interactive debugger for exercising a PMDA. See the **dbpmda(3)** man page for details.

In addition, careful use of PCP debugging flags can produce useful information concerning a PMDA's behavior; see the **PMAPI(3)** and **pmdbg(1)** man pages for a discussion of the PCP debugging and tracing framework.

13.7.2 Debugging Information

You can activate debugging options in PMCD and most other PCP tools with the **-D** command-line option. Supported options can be listed with the **pmdbg** command; see the **pmdbg(1)** man page. Setting the debug options for PMCD in `${PCP_PMCDOPTIONS_PATH}` might generate too much information to be useful, especially if there are other clients and PMDAs connected to the PMCD process.

The PMCD debugging options can also be changed dynamically by storing a new value into the metric **pmcd.control.debug**:

```
# pmstore pmcd.control.debug 5
```

Most of the **pcp_pmda** library functions log additional information if the **libpmda** option is set within the PMDA; see the **PMDA(3)** man page. The command-line argument **-D** is trapped by **pmdaGetOptions** to set the global debugging control options. Adding tests within the PMDA for the **appl0, appl1** and **appl2** trace flags permits different levels of information to be logged to the PMDA's log file.

All diagnostic, debugging, and tracing output from a PMDA should be written to the standard error stream.

Adding this segment of code to the **simple_store** metric causes a timestamped log message to be sent to the current log file whenever **pmstore** attempts to change **simple.numfetch** and the PCP debugging options have the **appl0** option set as shown in *Example 2.36. simple.numfetch in the Simple PMDA*:

Example 2.36. simple.numfetch in the Simple PMDA

```
case 0: /* simple.numfetch */
     x
     val = vsp->vlist[0].value.lval;
     if (val < 0) {
         sts = PM_ERR_SIGN;
         val = 0;
     }
     if (pmDebugOptions.appl0__) {
         pmNotifyErr(LOG_DEBUG,
               "simple: %d stored into numfetch", val);
     }
     numfetch = val;
     break;
```

For a description of **pmstore**, see the **pmstore(1)** man page.

13.7.3 dbpmda Debug Utility

The **dbpmda** utility provides a simple interface to the PDU communication protocol. It allows daemon and DSO PMDAs to be tested with most request types, while the PMDA process may be monitored with a debugger, tracing utilities, and other diagnostic tools. The **dbpmda(1)** man page contains a sample session with the **simple** PMDA.

13.8 Integration of a PMDA

Several steps are required to install (or remove) a PMDA from a production PMCD environment without affecting the operation of other PMDAs or related visualization and logging tools.

The PMDA typically would have its own directory below `${PCP_PMDAS_DIR}` into which several files would be installed. In the description in Section 2.8.1, "*Installing a PMDA*", the PMDA of interest is assumed to be known by the name **newbie**, hence the PMDA directory would be `${PCP_PMDAS_DIR}/newbie`.

Note: Any installation or removal of a PMDA involves updating files and directories that are typically well protected. Hence the procedures described in this section must be executed as the superuser.

13.8.1 Installing a PMDA

A PMDA is fully installed when these tasks are completed:

- Help text has been installed in a place where the PMDA can find it, usually in the PMDA directory `${PCP_PMDAS_DIR}/newbie`.
- The name space has been updated in the `${PCP_VAR_DIR}/pmns` directory.
- The PMDA binary has been installed, usually in the directory `${PCP_PMDAS_DIR}/newbie`.
- The `${PCP_PMCDCONF_PATH}` file has been updated.
- The PMCD process has been restarted or notified (with a **SIGHUP** signal) that the new PMDA exists.

The **Makefile** should include an **install** target to compile and link the PMDA (as a DSO, or a daemon or both) in the PMDA directory. The **clobber** target should remove any files created as a by-product of the **install** target.

You may wish to use `${PCP_PMDAS_DIR}/simple/Makefile` as a template for constructing a new PMDA **Makefile**; changing the assignment of **IAM** from **simple** to **newbie** would account for most of the required changes.

The **Install** script should make use of the generic procedures defined in the script `${PCP_SHARE_DIR}/lib/pmdaproc.sh`, and may be as straightforward as the one used for the trivial PMDA, shown in *Example 2.37. Install Script for the Trivial PMDA*:

Example 2.37. Install Script for the Trivial PMDA

```
. ${PCP_DIR}/etc/pcp.env
. ${PCP_SHARE_DIR}/lib/pmdaproc.sh

iam=trivial
pmdaSetup
pmdainstall
exit
```

The variables, shown in *Table 2.1. Variables to Control Behavior of Generic pmdaproc.sh Procedures*, may be assigned values to modify the behavior of the **pmdaSetup** and **pmdainstall** procedures from `${PCP_SHARE_DIR}/lib/pmdaproc.sh`.

Table 2.1. Variables to Control Behavior of Generic pmdaproc.sh Procedures

Shell Variable	Use	Default
$iam	Name of the PMDA; assignment to this variable is mandatory. Example: **iam=newbie**	
$dso_opt	Can this PMDA be installed as a DSO?	**false**
$daemon_opt	Can this PMDA be installed as a daemon?	**true**
$perl_opt	Is this PMDA a perl script?	**false**
$python_opt	Is this PMDA a python script?	**false**
$pipe_opt	If installed as a daemon PMDA, is the default IPC via pipes?	**true**
$socket_opt	If installed as a daemon PMDA, is the default IPC via an Internet socket?	**false**
$socket_inet_def	If installed as a daemon PMDA, and the IPC method uses an Internet socket, the default port number.	
$ipc_prot	IPC style for PDU exchanges involving a daemon PMDA; **binary** or **text**.	**binary**
$check_delay	Delay in seconds between installing PMDA and checking if metrics are available.	**3**
$args	Additional command-line arguments passed to a daemon PMDA.	
$pmns_source	The name of the PMNS file (by default relative to the PMDA directory).	**pmns**
$pmns_name	First-level name for this PMDA's metrics in the PMNS.	**$iam**
$help_source	The name of the help file (by default relative to the PMDA directory).	**help**
$pmda_name	The name of the executable for a daemon PMDA.	**pmda$iam**
$dso_name	The name of the shared library for a DSO PMDA.	**pmda$iam.$dso_suffix**
$dso_entry	The name of the initialization function for a DSO PMDA.	**${iam}_init**
$domain	The numerical PMDA domain number (from **domain.h**).	
$SYMDOM	The symbolic name of the PMDA domain number (from **domain.h**).	
$status	Exit status for the shell script	**0**

In addition, the variable **do_check** will be set to reflect the intention to check the availability of the metrics once the PMDA is installed. By default this variable is **true** however the command-line option **-Q** to **Install** may be used to set the variable to **false**.

Obviously, for anything but the most trivial PMDA, after calling the **pmdaSetup** procedure, the **Install** script should also prompt for any PMDA-specific parameters, which are typically accumulated in the *args* variable and used by the **pmdainstall** procedure.

The detailed operation of the **pmdainstall** procedure involves the following tasks:

- Using default assignments, and interaction where ambiguity exists, determine the PMDA type (DSO or daemon) and the IPC parameters, if any.
- Copy the `$pmns_source` file, replacing symbolic references to **SYMDOM** by the desired numeric domain number from **domain**.
- Merge the PMDA's name space into the PCP name space at the non-leaf node identified by **$pmns_name**.
- If any **pmchart** views can be found (files with names ending in ".pmchart"), copy these to the standard directory (`${PCP_VAR_DIR}/config/pmchart`) with the ".pmchart" suffix removed.
- Create new help files from `$help_source` after replacing symbolic references to **SYMDOM** by the desired numeric domain number from **domain**.
- Terminate the old daemon PMDA, if any.

- Use the **Makefile** to build the appropriate executables.
- Add the PMDA specification to PMCD's configuration file (`${PCP_PMCDCONF_PATH}`).
- Notify PMCD. To minimize the impact on the services PMCD provides, sending a **SIGHUP** to PMCD forces it to reread the configuration file and start, restart, or remove any PMDAs that have changed since the file was last read. However, if the newly installed PMDA must run using a different privilege level to PMCD then PMCD must be restarted. This is because PMCD runs unprivileged after initially starting the PMDAs.
- Check that the metrics from the new PMDA are available.

There are some PMDA changes that may trick PMCD into thinking nothing has changed, and not restarting the PMDA. Most notable are changes to the PMDA executable. In these cases, you may need to explicitly remove the PMDA as described in Section 2.8.2, "*Removing a PMDA*", or more drastically, restart PMCD as follows:

```
# ${PCP_RC_DIR}/pcp start
```

The files `${PCP_PMDAS_DIR}/*/Install` provide a wealth of examples that may be used to construct a new PMDA **Install** script.

13.8.2 Removing a PMDA

The simplest way to stop a PMDA from running, apart from killing the process, is to remove the entry from `${PCP_PMCDCONF_PATH}` and signal PMCD (with **SIGHUP**) to reread its configuration file. To completely remove a PMDA requires the reverse process of the installation, including an update of the Performance Metrics Name Space (PMNS).

This typically involves a **Remove** script in the PMDA directory that uses the same common procedures as the **Install** script described Section 2.8.1, "*Installing a PMDA*".

The `${PCP_PMDAS_DIR}/*/Remove` files provide a wealth of examples that may be used to construct a new PMDA **Remove** script.

13.8.3 Configuring PCP Tools

Most PCP tools have their own configuration file format for specifying which metrics to view or to log. By using canned configuration files that monitor key metrics of the new PMDA, users can quickly see the performance of the target system, as characterized by key metrics in the new PMDA.

Any configuration files that are created should be kept with the PMDA and installed into the appropriate directories when the PMDA is installed.

As with all PCP customization, some of the most valuable tools can be created by defining views, scenes, and control-panel layouts that combine related performance metrics from multiple PMDAs or multiple hosts.

Metrics suitable for on-going logging can be specified in templated **pmlogger** configuration files for **pmlogconf** to automatically add to the **pmlogger_daily** recorded set; see the **pmlogger(1), pmlogconf(1)** and **pmlogger_daily(1)** man pages.

Parameterized alarm configurations can be created using the **pmieconf** facilities; see the **pmieconf(1)** and **pmie(1)** man pages.

CHAPTER

FOURTEEN

PMAPI–THE PERFORMANCE METRICS API

Contents

- *PMAPI–The Performance Metrics API*
  - *Naming and Identifying Performance Metrics*
  - *Performance Metric Instances*
  - *Current PMAPI Context*
  - *Performance Metric Descriptions*
  - *Performance Metrics Values*
  - *Performance Event Metrics*
    - *Event Monitor Considerations*
    - *Event Collector Considerations*
  - *PMAPI Programming Style and Interaction*
    - *Variable Length Argument and Results Lists*
    - *Python Specific Issues*
    - *PMAPI Error Handling*
  - *PMAPI Procedural Interface*
    - *PMAPI Name Space Services*
      - *pmGetChildren Function*
      - *pmGetChildrenStatus Function*
      - *pmGetPMNSLocation Function*
      - *pmLoadNameSpace Function*
      - *pmLookupName Function*
      - *pmNameAll Function*
      - *pmNameID Function*
      - *pmTraversePMNS Function*
      - *pmUnloadNameSpace Function*
    - *PMAPI Metrics Description Services*

- pmLookupDesc Function
- pmLookupInDomText Function
- pmLookupText Function
- pmLookupLabels Function

* PMAPI Instance Domain Services
  - pmGetInDom Function
  - pmLookupInDom Function
  - pmNameInDom Function

* PMAPI Context Services
  - pmNewContext Function
  - pmDestroyContext Function
  - pmDupContext Function
  - pmUseContext Function
  - pmWhichContext Function
  - pmAddProfile Function
  - pmDelProfile Function
  - pmSetMode Function
  - pmReconnectContext Function
  - pmGetContextHostName Function

* PMAPI Timezone Services
  - pmNewContextZone Function
  - pmNewZone Function
  - pmUseZone Function
  - pmWhichZone Function

* PMAPI Metrics Services
  - pmFetch Function
  - pmFreeResult Function
  - pmStore Function

* PMAPI Fetchgroup Services
  - Fetchgroup setup
  - Fetchgroup operation
  - Fetchgroup shutdown

* PMAPI Record-Mode Services
  - pmRecordAddHost Function
  - pmRecordControl Function

            - *pmRecordSetup Function*
        * *PMAPI Archive-Specific Services*
            - *pmGetArchiveLabel Function*
            - *pmGetArchiveEnd Function*
            - *pmGetInDomArchive Function*
            - *pmLookupInDomArchive Function*
            - *pmNameInDomArchive Function*
            - *pmFetchArchive Function*
        * *PMAPI Time Control Services*
        * *PMAPI Ancillary Support Services*
            - *pmGetConfig Function*
            - *pmErrStr Function*
            - *pmExtractValue Function*
            - *pmConvScale Function*
            - *pmUnitsStr Function*
            - *pmIDStr Function*
            - *pmInDomStr Function*
            - *pmTypeStr Function*
            - *pmAtomStr Function*
            - *pmNumberStr Function*
            - *pmPrintValue Function*
            - *pmflush Function*
            - *pmprintf Function*
            - *pmSortInstances Function*
            - *pmParseInterval Function*
            - *pmParseMetricSpec Function*
    - *PMAPI Programming Issues and Examples*
        * *Symbolic Association between a Metric's Name and Value*
        * *Initializing New Metrics*
        * *Iterative Processing of Values*
        * *Accommodating Program Evolution*
        * *Handling PMAPI Errors*
        * *Compiling and Linking PMAPI Applications*

This chapter describes the Performance Metrics Application Programming Interface (PMAPI) provided with Performance Co-Pilot (PCP).

The PMAPI is a set of functions and data structure definitions that allow client applications to access performance data

from one or more Performance Metrics Collection Daemons (PMCDs) or from PCP archive logs. The PCP utilities are all written using the PMAPI.

The most common use of PCP includes running performance monitoring utilities on a workstation (the monitoring system) while performance data is retrieved from one or more remote collector systems by a number of PCP processes. These processes execute on both the monitoring system and the collector systems. The collector systems are typically servers, and are the targets for the performance investigations.

In the development of the PMAPI the most important question has been, "How easily and quickly will this API enable the user to build new performance tools, or exploit existing tools for newly available performance metrics?" The PMAPI and the standard tools that use the PMAPI have enjoyed a symbiotic evolution throughout the development of PCP.

It will be convenient to differentiate between code that uses the PMAPI and code that implements the services of the PMAPI. The former will be termed "above the PMAPI" and the latter "below the PMAPI."

14.1 Naming and Identifying Performance Metrics

Across all of the supported performance metric domains, there are a large number of performance metrics. Each metric has its own description, format, and semantics. PCP presents a uniform interface to these metrics above the PMAPI, independent of the source of the underlying metric data. For example, the performance metric **hinv.physmem** has a single 32-bit unsigned integer value, representing the number of megabytes of physical memory in the system, while the performance metric **disk.dev.total** has one 32-bit unsigned integer value per disk spindle, representing the cumulative count of I/O operations involving each associated disk spindle. These concepts are described in greater detail in Section 2.3, "*Domains, Metrics, Instances and Labels*".

For brevity and efficiency, internally PCP avoids using names for performance metrics, and instead uses an identification scheme that unambiguously associates a single integer with each known performance metric. This integer is known as a Performance Metric Identifier, or PMID. For functions using the PMAPI, a PMID is defined and manipulated with the typedef **pmID**.

Below the PMAPI, the integer value of the PMID has an internal structure that reflects the details of the PMCD and PMDA architecture, as described in Section 2.3.3, "*Metrics*".

Above the PMAPI, a Performance Metrics Name Space (PMNS) is used to provide a hierarchic classification of external metric names, and a one-to-one mapping of external names to internal PMIDs. A more detailed description of the PMNS can be found in the *Performance Co-Pilot User's and Administrator's Guide*.

The default PMNS comes from the performance metrics source, either a PMCD process or a PCP archive. This PMNS always reflects the available metrics from the performance metrics source.

14.2 Performance Metric Instances

When performance metric values are returned across the PMAPI to a requesting application, there may be more than one value for a particular metric; for example, independent counts for each CPU, or each process, or each disk, or each system call type, and so on. This multiplicity of values is not enumerated in the Name Space, but rather when performance metrics are delivered across the PMAPI.

The notion of **metric instances** is really a number of related concepts, as follows:

- A particular performance metric may have a set of associated values or instances.
- The instances are differentiated by an instance identifier.
- An instance identifier has an internal encoding (an integer value) and an external encoding (a corresponding external name or label).

- The set of all possible instance identifiers associated with a performance metric on a particular host constitutes an *instance domain*.
- Several performance metrics may share the same instance domain.

Consider *Example 3.1. Metrics Sharing the Same Instance Domain*:

Example 3.1. Metrics Sharing the Same Instance Domain

```
$ pminfo -f filesys.free

filesys.free
   inst [1 or "/dev/disk0"] value 1803
   inst [2 or "/dev/disk1"] value 22140
   inst [3 or "/dev/disk2"] value 157938
```

The metric **filesys.free** has three values, currently 1803, 22140, and 157938. These values are respectively associated with the instances identified by the internal identifiers 1, 2 and 3, and the external identifiers **/dev/disk0, /dev/disk1**, and **/dev/disk2**. These instances form an instance domain that is shared by the performance metrics **filesys.capacity, filesys.used, filesys.free, filesys.mountdir**, and so on.

Each performance metric is associated with an instance domain, while each instance domain may be associated with many performance metrics. Each instance domain is identified by a unique value, as defined by the following **typedef** declaration:

```
typedef unsigned long pmInDom;
```

The special instance domain **PM_INDOM_NULL** is reserved to indicate that the metric has a single value (a singular instance domain). For example, the performance metric **mem.freemem** always has exactly one value. Note that this is semantically different to a performance metric like **kernel.percpu.cpu.sys** that has a non-singular instance domain, but may have only one value available; for example, on a system with a single processor.

In the results returned above the PMAPI, each individual instance within an instance domain is identified by an internal integer instance identifier. The special instance identifier **PM_IN_NULL** is reserved for the single value in a singular instance domain. Performance metric values are delivered across the PMAPI as a set of instance identifier and value pairs.

The instance domain of a metric may change with time. For example, a machine may be shut down, have several disks added, and be rebooted. All performance metrics associated with the instance domain of disk devices would contain additional values after the reboot. The difficult issue of transient performance metrics means that repeated requests for the same PMID may return different numbers of values, or some changes in the particular instance identifiers returned. This means applications need to be aware that metric instantiation is guaranteed to be valid only at the time of collection.

Note: Some instance domains are more dynamic than others. For example, consider the instance domains behind the performance metrics **proc.memory.rss** (one instance per process), **swap.free** (one instance per swap partition) and **kernel.percpu.cpu.intr** (one instance per CPU).

14.3 Current PMAPI Context

When performance metrics are retrieved across the PMAPI, they are delivered in the context of a particular source of metrics, a point in time, and a profile of desired instances. This means that the application making the request has already negotiated across the PMAPI to establish the context in which the request should be executed.

A metric's source may be the current performance data from a particular host (a live or real-time source), or a set of archive logs of performance data collected by **pmlogger** at some remote host or earlier time (a retrospective or archive source). The metric's source is specified when the PMAPI context is created by calling the **pmNewContext** function. This function returns an opaque handle which can be used to identify the context.

The collection time for a performance metric is always the current time of day for a real-time source, or current position for an archive source. For archives, the collection time may be set to an arbitrary time within the bounds of the set of archive logs by calling the **pmSetMode** function.

The last component of a PMAPI context is an instance profile that may be used to control which particular instances from an instance domain should be retrieved. When a new PMAPI context is created, the initial state expresses an interest in all possible instances, to be collected at the current time. The instance profile can be manipulated using the **pmAddProfile** and **pmDelProfile** functions.

The current context can be changed by passing a context handle to **pmUseContext**. If a live context connection fails, the **pmReconnectContext** function can be used to attempt to reconnect it.

14.4 Performance Metric Descriptions

For each defined performance metric, there exists metadata describing it.

- A performance metric description (**pmDesc** structure) that describes the format and semantics of the performance metric.
- Help text associated with the metric and any associated instance domain.
- Performance metric labels (name:value pairs in **pmLabelSet** structures) associated with the metric and any associated instances.

The **pmDesc** structure, in *Example 3.2. pmDesc Structure*, provides all of the information required to interpret and manipulate a performance metric through the PMAPI. It has the following declaration:

Example 3.2. pmDesc Structure

```
/* Performance Metric Descriptor */
typedef struct {
    pmID    pmid;   /* unique identifier */
    int     type;   /* base data type (see below) */
    pmInDom indom;  /* instance domain */
    int     sem;    /* semantics of value (see below) */
    pmUnits units;  /* dimension and units (see below) */
} pmDesc;
```

The **type** field in the **pmDesc** structure describes various encodings of a metric's value. Its value will be one of the following constants:

```
/* pmDesc.type - data type of metric values */
#define PM_TYPE_NOSUPPORT -1   /* not in this version */
#define PM_TYPE_32        0    /* 32-bit signed integer */
#define PM_TYPE_U32       1    /* 32-bit unsigned integer */
#define PM_TYPE_64        2    /* 64-bit signed integer */
```

(continues on next page)

(continued from previous page)

```
#define PM_TYPE_U64       3    /* 64-bit unsigned integer */
#define PM_TYPE_FLOAT     4    /* 32-bit floating point */
#define PM_TYPE_DOUBLE    5    /* 64-bit floating point */
#define PM_TYPE_STRING    6    /* array of char */
#define PM_TYPE_AGGREGATE 7    /* arbitrary binary data */
#define PM_TYPE_AGGREGATE_STATIC 8 /* static pointer to aggregate */
#define PM_TYPE_EVENT     9    /* packed pmEventArray */
#define PM_TYPE_UNKNOWN   255  /* used in pmValueBlock not pmDesc */
```

By convention **PM_TYPE_STRING** is interpreted as a classic C-style null byte terminated string.

Event records are encoded as a packed array of strongly-typed, well-defined records within a **pmResult** structure, using a container metric with a value of type **PM_TYPE_EVENT**.

If the value of a performance metric is of type **PM_TYPE_STRING, PM_TYPE_AGGREGATE, PM_TYPE_AGGREGATE_STATIC**, or **PM_TYPE_EVENT**, the interpretation of that value is unknown to many PCP components. In the case of the aggregate types, the application using the value and the Performance Metrics Domain Agent (PMDA) providing the value must have some common understanding about how the value is structured and interpreted. Strings can be manipulated using the standard C libraries. Event records contain timestamps, event flags and event parameters, and the PMAPI provides support for unpacking an event record - see the **pmUnpackEventRecords(3)** man page for details. Further discussion on event metrics and event records can be found in Section 3.6, "*Performance Event Metrics*".

PM_TYPE_NOSUPPORT indicates that the PCP collection framework knows about the metric, but the corresponding service or application is either not configured or is at a revision level that does not provide support for this performance metric.

The semantics of the performance metric is described by the **sem** field of a **pmDesc** structure and uses the following constants:

```
/* pmDesc.sem - semantics of metric values */
#define PM_SEM_COUNTER  1  /* cumulative count, monotonic increasing */
#define PM_SEM_INSTANT  3  /* instantaneous value continuous domain */
#define PM_SEM_DISCRETE 4  /* instantaneous value discrete domain */
```

Each value for a performance metric is assumed to be drawn from a set of values that can be described in terms of their dimensionality and scale by a compact encoding, as follows:

- The dimensionality is defined by a power, or index, in each of three orthogonal dimensions: Space, Time, and Count (dimensionless). For example, I/O throughput is Space1.Time-1, while the running total of system calls is Count1, memory allocation is Space1, and average service time per event is Time1.Count-1.
- In each dimension, a number of common scale values are defined that may be used to better encode ranges that might otherwise exhaust the precision of a 32-bit value. For example, a metric with dimension Space1.Time-1 may have values encoded using the scale megabytes per second.

This information is encoded in the **pmUnits** data structure, shown in *Example 3.3. pmUnits and pmDesc Structures*. It is embedded in the pmDesc structure :

The structures are as follows:

Example 3.3. pmUnits and pmDesc Structures

```
/*
 * Encoding for the units (dimensions and
 * scale) for Performance Metric Values
 *
 * For example, a pmUnits struct of
```

(continues on next page)

(continued from previous page)

```
 * { 1, -1, 0, PM_SPACE_MBYTE, PM_TIME_SEC, 0 }
 * represents Mbytes/sec, while
 * { 0, 1, -1, 0, PM_TIME_HOUR, 6 }
 * represents hours/million-events
 */
typedef struct {
    int pad:8;
    int scaleCount:4; /* one of PM_COUNT_* below */
    int scaleTime:4;  /* one of PM_TIME_* below */
    int scaleSpace:4; /* one of PM_SPACE_* below */
    int dimCount:4;   /* event dimension */
    int dimTime:4;    /* time dimension */
    int dimSpace:4;   /* space dimension
} pmUnits;   /* dimensional units and scale of value */
/* pmUnits.scaleSpace */
#define PM_SPACE_BYTE 0  /* bytes */
#define PM_SPACE_KBYTE 1 /* kibibytes (1024) */
#define PM_SPACE_MBYTE 2 /* mebibytes (1024^2) */
#define PM_SPACE_GBYTE 3 /* gibibytes (1024^3) */
#define PM_SPACE_TBYTE 4 /* tebibytes (1024^4) */
#define PM_SPACE_PBYTE 5 /* pebibytes (1024^5) */
#define PM_SPACE_EBYTE 6 /* exbibytes (1024^6) */
#define PM_SPACE_ZBYTE 7 /* zebibytes (1024^7) */
#define PM_SPACE_YBYTE 8 /* yobibytes (1024^8) */
/* pmUnits.scaleTime */
#define PM_TIME_NSEC 0   /* nanoseconds */
#define PM_TIME_USEC 1   /* microseconds */
#define PM_TIME_MSEC 2   /* milliseconds */
#define PM_TIME_SEC 3    /* seconds */
#define PM_TIME_MIN 4    /* minutes */
#define PM_TIME_HOUR 5   /* hours */
/*
 * pmUnits.scaleCount (e.g. count events, syscalls,
 * interrupts, etc.) -- these are simply powers of 10,
 * and not enumerated here.
 * e.g. 6 for 10^6, or -3 for 10^-3
 */
#define PM_COUNT_ONE 0 /* 1 */
```

Metric and instance domain help text are simple ASCII strings. As a result, there are no special data structures associated with them. There are two forms of help text available for each metric and instance domain, however - one-line and long form.

Example 3.4. Help Text Flags

```
#define PM_TEXT_ONELINE 1
#define PM_TEXT_HELP    2
```

Labels are stored and communicated within PCP using JSONB formatted strings in the **json** field of a **pmLabelSet** structure. This format is a restricted form of JSON suitable for indexing and other operations. In JSONB form, insignificant whitespace is discarded, and order of label names is not preserved. Within the PMCS, however, a lexicographically sorted key space is always maintained. Duplicate label names are not permitted. The label with highest precedence in the label hierarchy (context level labels, domain level labels, and so on) is the only one presented.

Example 3.5. pmLabel and pmLabelSet Structures

```
typedef struct {
    uint     name : 16;      /* label name offset in JSONB string */
    uint     namelen : 8;    /* length of name excluding the null */
    uint     flags : 8;      /* information about this label */
    uint     value : 16;     /* offset of the label value */
    uint     valuelen : 16;  /* length of value in bytes */
} pmLabel;

/* flags identifying label hierarchy classes. */
#define PM_LABEL_CONTEXT        (1<<0)
#define PM_LABEL_DOMAIN         (1<<1)
#define PM_LABEL_INDOM          (1<<2)
#define PM_LABEL_CLUSTER        (1<<3)
#define PM_LABEL_ITEM           (1<<4)
#define PM_LABEL_INSTANCES      (1<<5)
/* flag identifying extrinsic labels. */
#define PM_LABEL_OPTIONAL       (1<<7)

typedef struct {
    uint     inst;          /* PM_IN_NULL or the instance ID */
    int      nlabels;       /* count of labels or error code */
    char     *json;         /* JSONB formatted labels string */
    uint     jsonlen : 16;  /* JSON string length byte count */
    uint     padding : 16;  /* zero, reserved for future use */
    pmLabel  *labels;       /* indexing into the JSON string */
} pmLabelSet;
```

The **pmLabel labels** array provides name and value indexes and lengths in the json string.

The **flags** field is a bitfield identifying the hierarchy level and whether this name:value pair is intrinsic (optional) or extrinsic (part of the mandatory, identifying metadata for the metric or instance). All other fields are offsets and lengths in the JSONB string from an associated **pmLabelSet** structure.

14.5 Performance Metrics Values

An application may fetch (or store) values for a set of performance metrics, each with a set of associated instances, using a single **pmFetch** (or **pmStore**) function call. To accommodate this, values are delivered across the PMAPI in the form of a tree data structure, rooted at a **pmResult** structure. This encoding is illustrated in *Figure 3.1. A Structured Result for Performance Metrics from pmFetch*, and uses the component data structures in *Example 3.6. pmValueBlock and pmValue Structures*:

Example 3.6. pmValueBlock and pmValue Structures

```
typedef struct {
    int inst;                /* instance identifier */
    union {
        pmValueBlock *pval;  /* pointer to value-block */
        int           lval;  /* integer value insitu */
    } value;
} pmValue;
```

Fig. 1: Figure 3.1. A Structured Result for Performance Metrics from pmFetch

The internal instance identifier is stored in the **inst** element. If a value for a particular metric-instance pair is a 32-bit

integer (signed or unsigned), then it will be stored in the **lval** element. If not, the value will be in a **pmValueBlock** structure, as shown in *Example 3.7. pmValueBlock Structure*, and will be located via **pval**:

The **pmValueBlock** structure is as follows:

Example 3.7. pmValueBlock Structure

```
typedef struct {
    unsigned int    vlen : 24;    /* bytes for vtype/vlen + vbuf */
    unsigned int    vtype : 8;    /* value type */
    char            vbuf[1];      /* the value */
} pmValueBlock;
```

The length of the **pmValueBlock** (including the **vtype** and **vlen** fields) is stored in **vlen**. Despite the prototype declaration of **vbuf**, this array really accommodates **vlen** minus **sizeof(vlen)** bytes. The **vtype** field encodes the type of the value in the **vbuf[]** array, and is one of the **PM_TYPE_*** macros defined in **<pcp/pmapi.h>**.

A **pmValueSet** structure, as shown in *Example 3.8. pmValueSet Structure*, contains all of the values to be returned from **pmFetch** for a single performance metric identified by the **pmid** field.

Example 3.8. pmValueSet Structure

```
typedef struct {
    pmID    pmid;          /* metric identifier */
    int     numval;        /* number of values */
    int     valfmt;        /* value style, insitu or ptr */
    pmValue vlist[1];      /* set of instances/values */
} pmValueSet;
```

If positive, the **numval** field identifies the number of value-instance pairs in the **vlist** array (despite the prototype declaration of size 1). If **numval** is zero, there are no values available for the associated performance metric and **vlist[0]** is undefined. A negative value for **numval** indicates an error condition (see the **pmErrStr(3)** man page) and **vlist[0]** is undefined. The **valfmt** field has the value **PM_VAL_INSITU** to indicate that the values for the performance metrics should be located directly via the **lval** member of the **value** union embedded in the elements of **vlist**; otherwise, metric values are located indirectly via the **pval** member of the elements of **vlist**.

The **pmResult** structure, as shown in *Example 3.9. pmResult Structure*, contains a time stamp and an array of **numpmid** pointers to **pmValueSet**.

Example 3.9. pmResult Structure

```
/* Result returned by pmFetch() */
typedef struct {
    struct timeval timestamp;    /* stamped by collector */
    int            numpmid;      /* number of PMIDs */
    pmValueSet     *vset[1];     /* set of value sets */
} pmResult
```

There is one **pmValueSet** pointer per PMID, with a one-to-one correspondence to the set of requested PMIDs passed to **pmFetch**.

Along with the metric values, the PMAPI returns a time stamp with each **pmResult** that serves to identify when the performance metric values were collected. The time is in the format returned by **gettimeofday** and is typically very close to the time when the metric values were extracted from their respective domains.

Note: There is a question of exactly when individual metrics may have been collected, especially given their origin in potentially different performance metric domains, and variability in metric updating frequency by individual PMDAs. PCP uses a pragmatic approach, in which the PMAPI implementation returns all metrics with values accurate as of the time stamp, to the maximum degree possible, and PMCD demands that all PMDAs deliver values within a small

realtime window. The resulting inaccuracy is small, and the additional burden of accurate individual timestamping for each returned metric value is neither warranted nor practical (from an implementation viewpoint).

The PMAPI provides functions to extract, rescale, and print values from the above structures; refer to Section 3.8.11, "*PMAPI Ancillary Support Services*".

14.6 Performance Event Metrics

In addition to performance metric values which are sampled by monitor tools, PCP supports the notion of performance event metrics which occur independently to any sampling frequency. These event metrics (PM_TYPE_EVENT) are delivered using a novel approach which allows both sampled and event trace data to be delivered via the same live wire protocol, the same on-disk archive format, and fundamentally using the same PMAPI services. In other words, a monitor tool may be sample and trace, simultaneously, using the PMAPI services discussed here.

Event metrics are characterised by certain key properties, distinguishing them from the other metric types (counters, instantaneous, and discrete):

- Occur at times outside of any monitor tools control, and often have a fine-grained timestamp associated with each event.
- Often have parameters associated with the event, which further describe each individual event, as shown in *Figure 3.2. Sample write(2) syscall entry point encoding*.
- May occur in very rapid succession, at rates such that both the collector and monitor sides may not be able to track all events. This property requires the PCP protocol to support the notion of "dropped" or "missed" events.
- There may be inherent relationships between events, for example the start and commit (or rollback) of a database transaction could be separate events, linked by a common transaction identifier (which would likely also be one of the parameters to each event). Begin-end and parent-child relationships are relatively common, and these properties require the PCP protocol to support the notion of "flags" that can be associated with events.

These differences aside, the representation of event metrics within PCP shares many aspects of the other metric types - event metrics appear in the Name Space (as do each of the event parameters), each has an associated Performance Metric Identifier and Descriptor, may have an instance domain associated with them, and may be recorded by **pmlogger** for subsequent replay.

Fig. 2: Figure 3.2. Sample write(2) syscall entry point encoding

Event metrics and their associated information (parameters, timestamps, flags, and so on) are delivered to monitoring tools alongside sampled metrics as part of the **pmResult** structure seen previously in *Example 3.9. pmResult Structure*.

The semantics of **pmFetch(3)** specifying an event metric PMID are such that all events observed on the collector since the previous fetch (by this specific monitor client) are to transferred to the monitor. Each event will have the metadata described earlier encoded with it (timestamps, flags, and so on) for each event. The encoding of the series of events involves a compound data structure within the **pmValueSet** associated with the event metric PMID, as illustrated in *Figure 3.3. Result Format for Event Performance Metrics from pmFetch*.

Fig. 3: Figure 3.3. Result Format for Event Performance Metrics from pmFetch

At the highest level, the "series of events" is encapsulated within a **pmEventArray** structure, as in *Example 3.10. pmEventArray and pmEventRecord Structures*:

Example 3.10. pmEventArray and pmEventRecord Structures

```
typedef struct {
    pmTimeval      er_timestamp;  /* 2 x 32-bit timestamp format */
    unsigned int     er_flags;      /* event record characteristics */
    int              er_nparams;    /* number of ea_param[] entries */
    pmEventParameter er_param[1];
} pmEventRecord;

typedef struct {
    unsigned int     ea_len :  24;  /* bytes for type/len + records */
    unsigned int     ea_type : 8;   /* value type */
    int              ea_nrecords;   /* number of ea_record entries */
    pmEventRecord    ea_record[1];
} pmEventArray;
```

Note that in the case of dropped events, the **pmEventRecord** structure is used to convey the number of events dropped - *er_flags* is used to indicate the presence of dropped events, and *er_nparams* is used to hold a count. Unsurprisingly, the parameters (*er_param*) will be empty in this situation.

The **pmEventParameter** structure is as follows:

Example 3.11. pmEventParameter Structure

```
typedef struct {
    pmID             ep_pmid;       /* parameter identifier */
    unsigned int     ep_type;       /* value type */
    int              ep_len;        /* bytes for type/len + vbuf */
    /* actual value (vbuf) here */
} pmEventParameter;
```

14.6.1 Event Monitor Considerations

In order to simplify the decoding of event record arrays, the PMAPI provides the **pmUnpackEventRecords** function for monitor tools. This function is passed a pointer to a **pmValueSet** associated with an event metric (within a **pmResult**) from a **pmFetch(3)**. For a given instance of that event metric, it returns an array of "unpacked" **pmResult** structures for each event.

The control information (flags and optionally dropped events) is included as derived metrics within each result structure. As such, these values can be queried similarly to other metrics, using their names - **event.flags** and **event.missed**. Note that these metrics will only exist after the first call to **pmUnpackEventRecords**.

An example of decoding event metrics in this way is presented in *Example 3.12. Unpacking Event Records from an Event Metric pmValueSet*:

Example 3.12. Unpacking Event Records from an Event Metric pmValueSet

```
enum { event_flags = 0, event_missed = 1 };
static char *metadata[] = { "event.flags", "event.missed" };
static pmID metapmid[2];

void dump_event(pmValueSet *vsp, int idx)
{
    pmResult    **res;
    int                 r, sts, nrecords;

    nrecords = pmUnpackEventRecords(vsp, idx, &res);
    if (nrecords < 0)
        fprintf(stderr, " pmUnpackEventRecords: %s\n", pmErrStr(nrecords));
```

(continues on next page)

(continued from previous page)

```
    else
        printf(" %d event records\n", nrecords);

    if ((sts = pmLookupName(2, &metadata, &metapmid)) < 0) {
        fprintf(stderr, "Event metadata error: %s\n", pmErrStr(sts));
        exit(1);
    }

    for (r = 0; r < nrecords; r++)
        dump_event_record(res, r);

    if (nrecords >= 0)
        pmFreeEventResult(res);
}

void dump_event_record(pmResult *res, int r)
{
    int         p;

    pmPrintStamp(stdout, &res[r]->timestamp);
    if (res[r]->numpmid == 0)
            printf(" ==> No parameters\n");
    for (p = 0; p < res[r]->numpmid; p++) {
        pmValueSet  *vsp = res[r]->vset[p];

        if (vsp->numval < 0) {
            int error = vsp->numval;
            printf("%s: %s\n", pmIDStr(vsp->pmid), pmErrStr(error));
        } else if (vsp->pmid == metapmid[event_flags]) {
            int flags = vsp->vlist[0].value.lval;
            printf(" flags 0x%x (%s)\n", flags, pmEventFlagsStr(flags));
        } else if (vsp->pmid == metapmid[event_missed]) {
            int count = vsp->vlist[0].value.lval;
            printf(" ==> %d missed event records\n", count);
        } else {
            dump_event_record_parameters(vsp);
        }
    }
}

void dump_event_record_parameters(pmValueSet *vsp)
{
    pmDesc      desc;
    char        *name;
    int         sts, j;

    if ((sts = pmLookupDesc(vsp->pmid, &desc)) < 0) {
        fprintf(stderr, "pmLookupDesc: %s\n", pmErrStr(sts));
    } else
    if ((sts = pmNameID(vsp->pmid, &name)) < 0) {
        fprintf(stderr, "pmNameID: %s\n", pmErrStr(sts));
    } else {
        printf("parameter %s", name);
        for (j = 0; j < vsp->numval; j++) {
            pmValue *vp = &vsp->vlist[j];
            if (vsp->numval > 1) {
                printf("[%d]", vp->inst);
```

(continues on next page)

(continued from previous page)

```
                pmPrintValue(stdout, vsp->valfmt, desc.type, vp, 1);
                putchar('\n');
            }
        }
        free(name);
    }
}
```

14.6.2 Event Collector Considerations

There is a feedback mechanism that is inherent in the design of the PCP monitor-collector event metric value exchange, which protects both monitor and collector components from becoming overrun by high frequency event arrivals. It is important that PMDA developers are aware of this mechanism and all of its implications.

Monitor tools can query new event arrival on whatever schedule they choose. There are no guarantees that this is a fixed interval, and no way for the PMDA to attempt to dictate this interval (nor should there be).

As a result, a PMDA that provides event metrics must:

- Track individual client connections using the per-client PMDA extensions (PMDA_INTERFACE_5 or later).
- Queue events, preferably in a memory-efficient manner, such that each interested monitor tool (there may be more than one) is informed of those events that arrived since their last request.
- Control the memory allocated to in-memory event storage. If monitors are requesting new events too slowly, compared to event arrival on the collector, the "missed events" feedback mechanism must be used to inform the monitor. This mechanism is also part of the model by which a PMDA can fix the amount of memory it uses. Once a fixed space is consumed, events can be dropped from the tail of the queue for each client, provided a counter is incremented and the client is subsequently informed.

Note: It is important that PMDAs are part of the performance solution, and not part of the performance problem! With event metrics, this is much more difficult to achieve than with counters or other sampled values.

There is certainly elegance to this approach for event metrics, and the way they dovetail with other, sampled performance metrics is unique to PCP. Notice also how the scheme naturally allows multiple monitor tools to consume the same events, no matter what the source of events is. The downside to this flexibility is increased complexity in the PMDA when event metrics are used.

This complexity comes in the form of event queueing and memory management, as well as per-client state tracking. Routines are available as part of the **pcp_pmda** library to assist, however - refer to the man page entries for **pmdaEventNewQueue(3)** and **pmdaEventNewClient(3)** for further details.

One final set of helper APIs is available to PMDA developers who incorporate event metrics. There is a need to build the **pmEventArray** structure, introduced in *Example 3.10. pmEventArray and pmEventRecord Structures*. This can be done directly, or using the helper routine **pmdaEventNewArray(3)**. If the latter, simpler model is chosen, the closely related routines **pmdaEventAddRecord, pmdaEventAddParam** and **pmdaEventAddMissedRecord** would also usually be used.

Depending on the nature of the events being exported by a PMDA, it can be desirable to perform **filtering** of events on the collector system. This reduces the amount of event traffic between monitor and collector systems (which may be filtered further on the monitor system, before presenting results). Some PMDAs have had success using the **pmStore(3)** mechanism to allow monitor tools to send a filter to the PMDA - using either a special control metric for the store operation, or the event metric itself. The filter sent will depend on the event metric, but it might be a regular expression, or a tracing script, or something else.

This technique has also been used to **enable** and **disable** event tracing entirely. It is often appropriate to make use of authentication and user credentials when providing such a facility (PMDA_INTERFACE_6 or later).

14.7 PMAPI Programming Style and Interaction

The following sections describe the PMAPI programming style:

- Variable length argument and results lists
- Python specific issues
- PMAPI error handling

14.7.1 Variable Length Argument and Results Lists

All arguments and results involving a "list of something" are encoded as an array with an associated argument or function value to identify the number of elements in the array. This encoding scheme avoids both the **varargs** approach and sentinel-terminated lists. Where the size of a result is known at the time of a call, it is the caller's responsibility to allocate (and possibly free) the storage, and the called function assumes that the resulting argument is of an appropriate size.

Where a result is of variable size and that size cannot be known in advance (for example, **pmGetChildren, pmGetInDom, pmNameInDom, pmNameID, pmLookupText**, **pmLookupLabels** and **pmFetch**), the underlying implementation uses dynamic allocation through **malloc** in the called function, with the caller responsible for subsequently calling **free** to release the storage when no longer required.

In the case of the result from **pmFetch**, there is a function (**pmFreeResult**) to release the storage, due to the complexity of the data structure and the need to make multiple calls to **free** in the correct sequence. Similarly, the **pmLookupLabels** function has an associated function (**pmFreeLabelSets**) to release the storage.

As a general rule, if the called function returns an error status, then no allocation is done, the pointer to the variable sized result is undefined, and **free**, **pmFreeLabelSets**, or **pmFreeResult** should not be called.

14.7.2 Python Specific Issues

A pcp client may be written in the python language by making use of the python bindings for PMAPI. The bindings use the python ctypes module to provide an interface to the PMAPI C language data structures. The primary imports that are needed by a client are:

- cpmapi which provides access to PMAPI constants

```
import cpmapi as c_api
```

- pmapi which provides access to PMAPI functions and data structures

```
from pcp import pmapi
```

- pmErr which provides access to the python bindings exception handler

```
from pcp.pmapi import pmErr
```

- pmgui which provides access to PMAPI record mode functions

```
from pcp import pmgui
```

Creating and destroying a PMAPI context in the python environment is done by creating and destroying an object of the pmapi class. This is done in one of two ways, either directly:

```
context = pmapi.pmContext()
```

or by automated processing of the command line arguments (refer to the **pmGetOptions** man page for greater detail).

```
options = pmapi.pmOptions(...)
context = pmapi.pmContext.fromOptions(options, sys.argv)
```

Most PMAPI C functions have python equivalents with similar, although not identical, call signatures. Some of the python functions do not return native python types, but instead return native C types wrapped by the ctypes library. In most cases these types are opaque, or nearly so; for example *pmid*:

```
pmid = context.pmLookupName("mem.freemem")
desc = context.pmLookupDescs(pmid)
result = context.pmFetch(pmid)
...
```

See the comparison of a standalone C and python client application in *Example 3.25. PMAPI Error Handling*.

14.7.3 PMAPI Error Handling

Where error conditions may arise, the functions that compose the PMAPI conform to a single, simple error notification scheme, as follows:

- The function returns an **int**. Values greater than or equal to zero indicate no error, and perhaps some positive status: for example, the number of items processed.
- Values less than zero indicate an error, as determined by a global table of error conditions and messages.

A PMAPI library function along the lines of **strerror** is provided to translate error conditions into error messages; see the **pmErrStr(3)** and **pmErrStr_r(3)** man pages. The error condition is returned as the function value from a previous PMAPI call; there is no global error indicator (unlike **errno**). This is to accommodate multi-threaded performance tools.

The available error codes may be displayed with the following command:

```
pmerr -l
```

Where possible, PMAPI routines are made as tolerant to failure as possible. In particular, routines which deal with compound data structures - results structures, multiple name lookups in one call and so on, will attempt to return all data that can be returned successfully, and errors embedded in the result where there were (partial) failures. In such cases a negative failure return code from the routine indicates catastrophic failure, otherwise success is returned and indicators for the partial failures are returned embedded in the results.

14.8 PMAPI Procedural Interface

The following sections describe all of the PMAPI functions that provide access to the PCP infrastructure on behalf of a client application:

- PMAPI Name Space services
- PMAPI metric description services
- PMAPI instance domain services

- PMAPI context services
- PMAPI timezone services
- PMAPI metrics services
- PMAPI fetchgroup services
- PMAPI record-mode services
- PMAPI archive-specific services
- PMAPI time control services
- PMAPI ancillary support services

14.8.1 PMAPI Name Space Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) Name Space services.

pmGetChildren Function

```
int pmGetChildren(const char*name, char***offspring)
Python:
[name1, name2...] = pmGetChildren(name)
```

Given a full pathname to a node in the current PMNS, as identified by *name*, return through *offspring* a list of the relative names of all the immediate descendents of *name* in the current PMNS. As a special case, if *name* is an empty string, (that is, **“”** but not **NULL** or **(char *)0**), the immediate descendents of the root node in the PMNS are returned.

For the python bindings a tuple containing the relative names of all the immediate descendents of *name* in the current PMNS is returned.

Normally, **pmGetChildren** returns the number of descendent names discovered, or a value less than zero for an error. The value zero indicates that the *name* is valid, and associated with a leaf node in the PMNS.

The resulting list of pointers (*offspring*) and the values (relative metric names) that the pointers reference are allocated by **pmGetChildren** with a single call to **malloc**, and it is the responsibility of the caller to issue a **free** (*offspring*) system call to release the space when it is no longer required. When the result of **pmGetChildren** is less than one, *offspring* is undefined (no space is allocated, and so calling **free** is counterproductive).

The python bindings return a tuple containing the relative names of all the immediate descendents of *name*, where *name* is a full pathname to a node in the current PMNS.

pmGetChildrenStatus Function

```
int pmGetChildrenStatus(const char *name, char ***offspring, int **status)
Python:
([name1, name2...],[status1, status2...]) = pmGetChildrenStatus(name)
```

The **pmGetChildrenStatus** function is an extension of **pmGetChildren** that optionally returns status information about each of the descendent names.

Given a fully qualified pathname to a node in the current PMNS, as identified by *name*, **pmGetChildrenStatus** returns by means of *offspring* a list of the relative names of all of the immediate descendent nodes of *name* in the current PMNS. If *name* is the empty string (“”), it returns the immediate descendents of the root node in the PMNS.

If *status* is not NULL, then **pmGetChildrenStatus** also returns the status of each child by means of *status*. This refers to either a leaf node (with value **PMNS_LEAF_STATUS**) or a non-leaf node (with value **PMNS_NONLEAF_STATUS**).

Normally, **pmGetChildrenStatus** returns the number of descendent names discovered, or else a value less than zero to indicate an error. The value zero indicates that name is a valid metric name, being associated with a leaf node in the PMNS.

The resulting list of pointers (*offspring*) and the values (relative metric names) that the pointers reference are allocated by **pmGetChildrenStatus** with a single call to **malloc**, and it is the responsibility of the caller to **free** (*offspring*) to release the space when it is no longer required. The same holds true for the *status* array.

The python bindings return a tuple containing the relative names and statuses of all the immediate descendents of *name*, where *name* is a full pathname to a node in the current PMNS.

pmGetPMNSLocation Function

```
int pmGetPMNSLocation(void)
Python:
int loc = pmGetPMNSLocation()
```

If an application needs to know where the origin of a PMNS is, **pmGetPMNSLocation** returns whether it is an archive (**PMNS_ARCHIVE**), a local PMNS file (**PMNS_LOCAL**), or a remote PMCD (**PMNS_REMOTE**). This information may be useful in determining an appropriate error message depending on PMNS location.

The python bindings return whether a PMNS is an archive *cpmapi.PMNS_ARCHIVE*, a local PMNS file *cpmapi.PMNS_LOCAL*, or a remote PMCD *cpmapi.PMNS_REMOTE*. The constants are available by importing cpmapi.

pmLoadNameSpace Function

```
int pmLoadNameSpace(const char *filename)
Python:
int status = pmLoadNameSpace(filename)
```

In the highly unusual situation that an application wants to force using a local Performance Metrics Name Space (PMNS), the application can load the PMNS using **pmLoadNameSpace**.

The *filename* argument designates the PMNS of interest. For applications that do not require a tailored Name Space, the special value **PM_NS_DEFAULT** may be used for *filename*, to force a default local PMNS to be established. Externally, a PMNS is stored in an ASCII format.

The python bindings load a local tailored Name Space from *filename*.

Note: Do not use this routine in monitor tools. The distributed PMNS services avoid the need for a local PMNS; so applications should **not** use **pmLoadNameSpace**. Without this call, the default PMNS is the one at the source of the performance metrics (PMCD or an archive).

---

pmLookupName Function

```
int pmLookupName(int numpmid, char *namelist[], pmID pmidlist[])
Python:
c_uint pmid [] = pmLookupName("MetricName")
c_uint pmid [] = pmLookupName(("MetricName1", "MetricName2", ...))
```

Given a list in *namelist* containing *numpmid* full pathnames for performance metrics from the current PMNS, **pmLookupName** returns the list of associated PMIDs through the *pmidlist* parameter. Invalid metrics names are translated to the error PMID value of **PM_ID_NULL**.

The result from **pmLookupName** is the number of names translated in the absence of errors, or an error indication. Note that argument definition and the error protocol guarantee a one-to-one relationship between the elements of *namelist* and *pmidlist*; both lists contain exactly *numpmid* elements.

The python bindings return an array of associated PMIDs corresponding to a tuple of *MetricNames*. The returned *pmid* tuple is passed to **pmLookupDescs** and **pmFetch**.

pmNameAll Function

```
int pmNameAll(pmID pmid, char ***nameset)
Python:
[name1, name2...] = pmNameAll(pmid)
```

Given a performance metric ID in *pmid*, **pmNameAll** determines all the corresponding metric names, if any, in the PMNS, and returns these through *nameset*.

The resulting list of pointers *nameset* and the values (relative names) that the pointers reference are allocated by **pmNameAll** with a single call to **malloc**. It is the caller's responsibility to call **free** and release the space when it is no longer required.

In the absence of errors, **pmNameAll** returns the number of names in **nameset**.

For many PMNS instances, there is a 1:1 mapping between a name and a PMID, and under these circumstances, **pmNameID** provides a simpler interface in the absence of duplicate names for a particular PMID.

The python bindings return a tuple of all metric names having this identical *pmid*.

pmNameID Function

```
int pmNameID(pmID pmid, char **name)
Python:
"metric name" = pmNameID(pmid)
```

Given a performance metric ID in *pmid*, **pmNameID** determines the corresponding metric name, if any, in the current PMNS, and returns this through *name*.

In the absence of errors, **pmNameID** returns zero. The *name* argument is a null byte terminated string, allocated by **pmNameID** using **malloc**. It is the caller's responsibility to call **free** and release the space when it is no longer required.

The python bindings return a metric name corresponding to a *pmid*.

pmTraversePMNS Function

```
int pmTraversePMNS(const char *name, void (*dometric)(const char *))
Python:
int status = pmTraversePMNS(name, traverse_callback)
```

The function **pmTraversePMNS** may be used to perform a depth-first traversal of the PMNS. The traversal starts at the node identified by *name* –if *name* is an empty string, the traversal starts at the root of the PMNS. Usually, *name* would be the pathname of a non-leaf node in the PMNS.

For each leaf node (actual performance metrics) found in the traversal, the user-supplied function **dometric** is called with the full pathname of that metric in the PMNS as the single argument; this argument is a null byte-terminated string, and is constructed from a buffer that is managed internally to **pmTraversePMNS**. Consequently, the value is valid only during the call to **dometric**–if the pathname needs to be retained, it should be copied using **strdup** before returning from **dometric**; see the **strdup(3)** man page.

The python bindings perform a depth first traversal of the PMNS by scanning *namespace*, depth first, and call a python function *traverse_callback* for each node.

pmUnloadNameSpace Function

```
int pmUnloadNameSpace(void)
Python:
pmUnLoadNameSpace("NameSpace")
```

If a local PMNS was loaded with **pmLoadNameSpace**, calling **pmUnloadNameSpace** frees up the memory associated with the PMNS and force all subsequent Name Space functions to use the distributed PMNS. If **pmUnloadNameSpace** is called before calling **pmLoadNameSpace**, it has no effect.

As discussed in Section 3.8.1.4, "*pmLoadNameSpace Function*" there are few if any situations where clients need to call this routine in modern versions of PCP.

14.8.2 PMAPI Metrics Description Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) metric description services.

pmLookupDesc Function

```
int pmLookupDesc(pmID pmid, pmDesc *desc)
Python:
pmDesc* pmdesc = pmLookupDesc(c_uint pmid)
(pmDesc* pmdesc)[] = pmLookupDescs(c_uint pmids[N])
(pmDesc* pmdesc)[] = pmLookupDescs(c_uint pmid)
```

Given a Performance Metric Identifier (PMID) as *pmid*, **pmLookupDesc** returns the associated **pmDesc** structure through the parameter *desc* from the current PMAPI context. For more information about **pmDesc**, see Section 3.4, "*Performance Metric Descriptions*".

The python bindings return the metric description structure **pmDesc** corresponding to *pmid*. The returned *pmdesc* is passed to **pmExtractValue** and **pmLookupInDom**. The python bindings provide an entry **pmLookupDescs** that is similar to pmLookupDesc but does a metric description lookup for each element in a PMID array *pmids*.

pmLookupInDomText Function

```
int pmLookupInDomText(pmInDom indom, int level, char **buffer)
Python:
"metric description" = pmGetInDomText(pmDesc pmdesc)
```

Provided the source of metrics from the current PMAPI context is a host, retrieve descriptive text about the performance metrics instance domain identified by *indom*.

The *level* argument should be **PM_TEXT_ONELINE** for a one-line summary, or **PM_TEXT_HELP** for a more verbose description suited to a help dialogue. The space pointed to by *buffer* is allocated in **pmLookupInDomText** with **malloc**, and it is the responsibility of the caller to free unneeded space; see the **malloc(3)** and **free(3)** man pages.

The help text files used to implement **pmLookupInDomText** are often created using **newhelp** and accessed by the appropriate PMDA response to requests forwarded to the PMDA by PMCD. Further details may be found in Section 2.4.4, "*PMDA Help Text*" .

The python bindings lookup the description text about the performance metrics pmDesc *pmdesc*. The default is a one line summary; for a more verbose description add an optional second parameter *cpmapi.PM_TEXT_HELP*. The constant is available by importing cpmapi.

pmLookupText Function

```
int pmLookupText(pmID pmid, int level, char **buffer)
Python:
"metric description" = pmLookupText(c_uint pmid)
```

Retrieve descriptive text about the performance metric identified by *pmid*. The argument *level* should be **PM_TEXT_ONELINE** for a one-line summary, or **PM_TEXT_HELP** for a more verbose description, suited to a help dialogue.

The space pointed to by *buffer* is allocated in **pmLookupText** with **malloc**, and it is the responsibility of the caller to **free** the space when it is no longer required; see the **malloc(3)** and **free(3)** man pages.

The help text files used to implement **pmLookupText** are created using **newhelp** and accessed by the appropriate PMDA in response to requests forwarded to the PMDA by PMCD. Further details may be found in Section 2.4.4, "*PMDA Help Text*".

The python bindings lookup the description text about the performance metrics pmID *pmid*. The default is a one line summary; for a more verbose description add an optional second parameter *cpmapi.PM_TEXT_HELP*. The constant is available by importing cpmapi.

pmLookupLabels Function

```
int pmLookupLabels(pmID pmid, pmLabelSet **labelsets)
Python:
(pmLabelSet* pmlabelset)[] pmLookupLabels(c_uint pmid)
```

Retrieve **name:value** pairs providing additional identity and descriptive metadata about the performance metric identified by *pmid*.

The space pointed to by *labelsets* is allocated in **pmLookupLabels** with potentially multiple calls to **malloc** and it is the responsibility of the caller to **pmFreeLabelSets** the space when it is no longer required; see the **malloc(3)** and **pmFreeLabelSets(3)** man pages.

Additional helper interfaces are also available, used internally by **pmLookupLabels** and to help with post-processing of *labelsets*. See the **pmLookupLabels(3)** and **pmMergeLabelSets(3)** man pages.

14.8.3 PMAPI Instance Domain Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) instance domain services.

pmGetInDom Function

```
int pmGetInDom(pmInDom indom, int **instlist, char ***namelist)
Python:
([instance1, instance2...] [name1, name2...]) pmGetInDom(pmDesc pmdesc)
```

In the current PMAPI context, locate the description of the instance domain *indom*, and return through *instlist* the internal instance identifiers for all instances, and through *namelist* the full external identifiers for all instances. The number of instances found is returned as the function value (or less than zero to indicate an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by **pmGetInDom** with two calls to **malloc**, and it is the responsibility of the caller to use **free** (*instlist*) and **free** (*namelist*) to release the space when it is no longer required. When the result of **pmGetInDom** is less than one, both *instlist* and *namelist* are undefined (no space is allocated, and so calling **free** is a bad idea); see the **malloc(3)** and **free(3)** man pages.

The python bindings return a tuple of the instance identifiers and instance names for an instance domain *pmdesc*.

pmLookupInDom Function

```
int pmLookupInDom(pmInDom indom, const char *name)
Python:
int instid = pmLookupInDom(pmDesc pmdesc, "Instance")
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the external identification given by *name*, and return the internal instance identifier.

The python bindings return the instance id corresponding to *"Instance"* in the instance domain *pmdesc*.

pmNameInDom Function

```
int pmNameInDom(pmInDom indom, int inst, char **name)
Python:
"instance id" = pmNameInDom(pmDesc pmdesc, c_uint instid)
```

For the instance domain *indom*, in the current PMAPI context, locate the instance with the internal instance identifier given by *inst*, and return the full external identification through *name*. The space for the value of *name* is allocated in **pmNameInDom** with **malloc**, and it is the responsibility of the caller to free the space when it is no longer required; see the **malloc(3)** and **free(3)** man pages.

The python bindings return the text name of an instance corresponding to an instance domain *pmdesc* with instance identifier *instid*.

14.8.4 PMAPI Context Services

Table 3.1. Context Components of PMAPI Functions shows which of the three components of a PMAPI context (metrics source, instance profile, and collection time) are relevant for various PMAPI functions. Those PMAPI functions not shown in this table either manipulate the PMAPI context directly, or are executed independently of the current PMAPI context.

Table 3.1. Context Components of PMAPI Functions

Function name	Metrics Source	Instance Profile	Collection Time	Notes
pmAddProfile	Yes	Yes		
pmDelProfile	Yes	Yes		
pmDupContext	Yes	Yes	Yes	
pmFetch	Yes	Yes	Yes	
pmFetchArchive	Yes		Yes	(1)
pmGetArchiveEnd	Yes			(1)
pmGetArchiveLabel	Yes			(1)
pmGetChildren	Yes			
pmGetChildrenStatus	Yes			
pmGetContextHostName	Yes			
pmGetPMNSLocation	Yes			
pmGetInDom	Yes		Yes	(2)
pmGetInDomArchive	Yes			(1)
pmLookupDesc	Yes			(3)
pmLookupInDom	Yes		Yes	(2)
pmLookupInDomArchive	Yes			(1,2)
pmLookupInDomText	Yes			
pmLookupLabels	Yes			
pmLookupName	Yes			
pmLookupText	Yes			
pmNameAll	Yes			
pmNameID	Yes			
pmNameInDom	Yes		Yes	(2)
pmNameInDomArchive	Yes			(1,2)
pmSetMode	Yes		Yes	
pmStore	Yes			(4)
pmTraversePMNS	Yes			

Notes:

1. Operation supported only for PMAPI contexts where the source of metrics is an archive.
2. A specific instance domain is included in the arguments to these functions, and the result is independent of the instance profile for any PMAPI context.
3. The metadata that describes a performance metric is sensitive to the source of the metrics, but independent of any instance profile and of the collection time.
4. This operation is supported only for contexts where the source of the metrics is a host. Further, the instance identifiers are included in the argument to the function, and the effects upon the current values of the metrics are immediate (retrospective changes are not allowed). Consequently, from the current PMAPI context, neither the instance profile nor the collection time influence the result of this function.

pmNewContext Function

```
int pmNewContext(int type, const char *name)
```

The **pmNewContext** function may be used to establish a new PMAPI context. The source of metrics is identified by *name*, and may be a host specification (*type* is **PM_CONTEXT_HOST**) or a comma-separated list of names referring to a set of archive logs (*type* is **PM_CONTEXT_ARCHIVE**). Each element of the list may either be the base name common to all of the physical files of an archive log or the name of a directory containing archive logs.

A host specification usually contains a simple hostname, an internet address (IPv4 or IPv6), or the path to the PMCD Unix domain socket. It can also specify properties of the connection to PMCD, such as the protocol to use (secure and encrypted, or native) and whether PMCD should be reached via a **pmproxy** host. Various other connection attributes, such as authentication information (user name, password, authentication method, and so on) can also be specified. Further details can be found in the **PCPIntro(3)** man page, and the companion *Performance Co-Pilot Tutorials and Case Studies* document.

In the case where *type* is **PM_CONTEXT_ARCHIVE**, there are some restrictions on the archives within the specified set:

- The archives must all have been generated on the same host.
- The archives must not overlap in time.
- The archives must all have been created using the same time zone.
- The pmID of each metric should be the same in all of the archives. Multiple pmIDs are currently tolerated by using the first pmID defined for each metric and ignoring subsequent pmIDs.
- The type of each metric must be the same in all of the archives.
- The semantics of each metric must be the same in all of the archives.
- The units of each metric must be the same in all of the archives.
- The instance domain of each metric must be the same in all of the archives.

In the case where *type* is **PM_CONTEXT_LOCAL**, *name* is ignored, and the context uses a stand-alone connection to the PMDA methods used by PMCD. When this type of context is in effect, the range of accessible performance metrics is constrained to DSO PMDAs listed in the **pmcd** configuration file `${PCP_PMCDCONF_PATH}`. The reason this is done, as opposed to all of the DSO PMDAs found below `${PCP_PMDAS_DIR}` for example, is that DSO PMDAs listed there are very likely to have their metric names reflected in the local Name Space file, which will be loaded for this class of context.

The initial instance profile is set up to select all instances in all instance domains, and the initial collection time is the current time at the time of each request for a host, or the time at the start of the first log for a set of archives. In the case of archives, the initial collection time results in the earliest set of metrics being returned from the set of archives at the first **pmFetch**.

Once established, the association between a PMAPI context and a source of metrics is fixed for the life of the context; however, functions are provided to independently manipulate both the instance profile and the collection time components of a context.

The function returns a "handle" that may be used in subsequent calls to **pmUseContext**. This new PMAPI context stays in effect for all subsequent context sensitive calls across the PMAPI until another call to **pmNewContext** is made, or the context is explicitly changed with a call to **pmDupContext** or **pmUseContext**.

For the python bindings creating and destroying a PMAPI context is done by creating and destroying an object of the pmapi class.

pmDestroyContext Function

```
int pmDestroyContext(int handle)
```

The PMAPI context identified by *handle* is destroyed. Typically, this implies terminating a connection to PMCD or closing an archive file, and orderly clean-up. The PMAPI context must have been previously created using **pmNewContext** or **pmDupContext**.

On success, **pmDestroyContext** returns zero. If *handle* was the current PMAPI context, then the current context becomes undefined. This means the application must explicitly re-establish a valid PMAPI context with **pmUseContext**, or create a new context with **pmNewContext** or **pmDupContext**, before the next PMAPI operation requiring a PMAPI context.

For the python bindings creating and destroying a PMAPI context is done by creating and destroying an object of the pmapi class.

pmDupContext Function

```
int pmDupContext(void)
```

Replicate the current PMAPI context (source, instance profile, and collection time). This function returns a handle for the new context, which may be used with subsequent calls to **pmUseContext**. The newly replicated PMAPI context becomes the current context.

pmUseContext Function

```
int pmUseContext(int handle)
```

Calling **pmUseContext** causes the current PMAPI context to be set to the context identified by *handle*. The value of *handle* must be one returned from an earlier call to **pmNewContext** or **pmDupContext**.

Below the PMAPI, all contexts used by an application are saved in their most recently modified state, so **pmUseContext** restores the context to the state it was in the last time the context was used, not the state of the context when it was established.

pmWhichContext Function

```
int pmWhichContext(void)
Python:
int ctx_idx = pmWhichContext()
```

Returns the handle for the current PMAPI context (source, instance profile, and collection time).

The python bindings return the handle of the current PMAPI context.

pmAddProfile Function

```
int pmAddProfile(pmInDom indom, int numinst, int instlist[])
Python:
int status = pmAddProfile(pmDesc pmdesc, [c_uint instid])
```

Add new instance specifications to the instance profile of the current PMAPI context. At its simplest, instances identified by the *instlist* argument for the *indom* instance domain are added to the instance profile. The list of instance identifiers contains *numinst* values.

If *indom* equals **PM_INDOM_NULL**, or *numinst* is zero, then all instance domains are selected. If *instlist* is NULL, then all instances are selected. To enable all available instances in all domains, use this syntax:

```
pmAddProfile(PM_INDOM_NULL, 0, NULL).
```

The python bindings add the list of instances *instid* to the instance profile of the instance *pmdesc*.

pmDelProfile Function

```
int pmDelProfile(pmInDom indom, int numinst, int instlist[])
Python:
int status = pmDelProfile(pmDesc pmdesc, c_uint instid)
int status = pmDelProfile(pmDesc pmdesc, [c_uint instid])
```

Delete instance specifications from the instance profile of the current PMAPI context. In the simplest variant, the list of instances identified by the *instlist* argument for the *indom* instance domain is removed from the instance profile. The list of instance identifiers contains *numinst* values.

If *indom* equals **PM_INDOM_NULL**, then all instance domains are selected for deletion. If *instlist* is NULL, then all instances in the selected domains are removed from the profile. To disable all available instances in all domains, use this syntax:

```
pmDelProfile(PM_INDOM_NULL, 0, NULL)
```

The python bindings delete the list of instances *instid* from the instance profile of the instance domain *pmdesc*.

pmSetMode Function

```
int pmSetMode(int mode, const struct timeval *when, int delta)
Python:
int status = pmSetMode(mode, timeVal timeval, int delta)
```

This function defines the collection time and mode for accessing performance metrics and metadata in the current PMAPI context. This mode affects the semantics of subsequent calls to the following PMAPI functions: **pmFetch, pmFetchArchive, pmLookupDesc, pmGetInDom, pmLookupInDom** , and **pmNameInDom**.

The **pmSetMode** function requires the current PMAPI context to be of type **PM_CONTEXT_ARCHIVE**.

The *when* parameter defines a time origin, and all requests for metadata (metrics descriptions and instance identifiers from the instance domains) are processed to reflect the state of the metadata as of the time origin. For example, use the last state of this information at, or before, the time origin.

If the *mode* is **PM_MODE_INTERP** then, in the case of **pmFetch**, the underlying code uses an interpolation scheme to compute the values of the metrics from the values recorded for times in the proximity of the time origin.

If the *mode* is **PM_MODE_FORW**, then, in the case of **pmFetch**, the collection of recorded metric values is scanned forward, until values for at least one of the requested metrics is located after the time origin. Then all requested metrics stored in the PCP archive at that time are returned with a corresponding time stamp. This is the default mode when an archive context is first established with **pmNewContext**.

If the *mode* is **PM_MODE_BACK**, then the situation is the same as for **PM_MODE_FORW**, except a **pmFetch** is serviced by scanning the collection of recorded metrics backward for metrics before the time origin.

After each successful **pmFetch**, the time origin is reset to the time stamp returned through the **pmResult**.

The **pmSetMode** parameter *delta* defines an additional number of time unit that should be used to adjust the time origin (forward or backward) after the new time origin from the **pmResult** has been determined. This is useful when moving through archives with a mode of **PM_MODE_INTERP**. The high-order bits of the *mode* parameter field is also used to optionally set the units of time for the **delta** field. To specify the units of time, use the **PM_XTB_SET** macro with one of the values **PM_TIME_NSEC, PM_TIME_MSEC, PM_TIME_SEC**, or so on as follows:

```
PM_MODE_INTERP | PM_XTB_SET(PM_TIME_XXXX)
```

If no units are specified, the default is to interpret *delta* as milliseconds.

Using these mode options, an application can implement replay, playback, fast forward, or reverse for performance metric values held in a set of PCP archive logs by alternating calls to **pmSetMode** and **pmFetch**.

In *Example 3.13. Dumping Values in Temporal Sequence*, the code fragment may be used to dump only those values stored in correct temporal sequence, for the specified performance metric **my.metric.name**:

Example 3.13. Dumping Values in Temporal Sequence

```
int     sts;
pmID    pmid;
char    *name = "my.metric.name";

sts = pmNewContext(PM_CONTEXT_ARCHIVE, "myarchive");
sts = pmLookupName(1, &name, &pmid);
for ( ; ; ) {
    sts = pmFetch(1, &pmid, &result);
    if (sts < 0)
        break;
    /* dump value(s) from result->vset[0]->vlist[] */
    pmFreeResult(result);
}
```

Alternatively, the code fragment in *Example 3.14. Replaying Interpolated Metrics* may be used to replay interpolated metrics from an archive in reverse chronological order, at ten-second intervals (of recorded time):

Example 3.14. Replaying Interpolated Metrics

```
int             sts;
pmID            pmid;
char            *name = "my.metric.name";
struct timeval  endtime;

sts = pmNewContext(PM_CONTEXT_ARCHIVE, "myarchive");
sts = pmLookupName(1, &name, &pmid);
sts = pmGetArchiveEnd(&endtime);
sts = pmSetMode(PM_MODE_INTERP, &endtime, -10000);
while (pmFetch(1, &pmid, &result) != PM_ERR_EOL) {
    /*
     * process interpolated metric values as of result->timestamp
```

(continues on next page)

(continued from previous page)

```
     */
    pmFreeResult(result);
}
```

The python bindings define the collection *time* and *mode* for reading archive files. *mode* can be one of: c_api.PM_MODE_LIVE, c_api.PM_MODE_INTERP, c_api.FORW, c_api.BACK. wjocj are available by importing cpmapi.

pmReconnectContext Function

```
int pmReconnectContext(int handle)
Python:
int status = pmReconnectContext()
```

As a result of network, host, or PMCD (Performance Metrics Collection Daemon) failure, an application's connection to PMCD may be established and then lost.

The function **pmReconnectContext** allows an application to request that the PMAPI context identified by *handle* be re-established, provided the associated PMCD is accessible.

Note: *handle* may or may not be the current context.

To avoid flooding the system with reconnect requests, **pmReconnectContext** attempts a reconnection only after a suitable delay from the previous attempt. This imposed restriction on the reconnect re-try time interval uses a default exponential back-off so that the initial delay is 5 seconds after the first unsuccessful attempt, then 10 seconds, then 20 seconds, then 40 seconds, and then 80 seconds thereafter. The intervals between reconnection attempts may be modified using the environment variable **PMCD_RECONNECT_TIMEOUT** and the time to wait before an attempted connection is deemed to have failed is controlled by the **PMCD_CONNECT_TIMEOUT** environment variable; see the **PCPIntro(1)** man page.

If the reconnection succeeds, **pmReconnectContext** returns *handle*. Note that even in the case of a successful reconnection, **pmReconnectContext** does not change the current PMAPI context.

The python bindings reestablish the connection for the context.

pmGetContextHostName Function

```
const char *pmGetContextHostName(int id)
char *pmGetContextHostName_r(int id, char *buf, int buflen)
Python:
"hostname" = pmGetContextHostName()
```

Given a valid PCP context identifier previously created with **pmNewContext** or **pmDupContext**, the **pmGetContextHostName** function provides a possibility to retrieve a host name associated with a context regardless of the context type.

This function will use the **pmcd.hostname** metric if it is available, and so is able to provide an accurate hostname in the presence of connection tunnelling and port forwarding.

If *id* is not a valid PCP context identifier, this function returns a zero length string and therefore never fails.

In the case of **pmGetContextHostName**, the string value is held in a single static buffer, so concurrent calls may not produce the desired results. The **pmGetContextHostName_r** function allows a buffer and length to be passed in, into which the message is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings query the current context hostname.

14.8.5 PMAPI Timezone Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) timezone services.

pmNewContextZone Function

```
int pmNewContextZone(void)
Python:
pmNewContextZone()
```

If the current PMAPI context is an archive, the **pmNewContextZone** function uses the timezone from the archive label record in the first archive of the set to set the current reporting timezone. The current reporting timezone affects the timezone used by **pmCtime** and **pmLocaltime**.

If the current PMAPI context corresponds to a host source of metrics, **pmNewContextZone** executes a **pmFetch** to retrieve the value for the metric **pmcd.timezone** and uses that to set the current reporting timezone.

In both cases, the function returns a value to identify the current reporting timezone that may be used in a subsequent call to **pmUseZone** to restore this reporting timezone.

PM_ERR_NOCONTEXT indicates the current PMAPI context is not valid. A return value less than zero indicates a fatal error from a system call, most likely **malloc**.

pmNewZone Function

```
int pmNewZone(const char *tz)
Python:
int tz_handle = pmNewZone(int tz)
```

The **pmNewZone** function sets the current reporting timezone, and returns a value that may be used in a subsequent call to **pmUseZone** to restore this reporting timezone. The current reporting timezone affects the timezone used by **pmCtime** and **pmLocaltime**.

The *tz* argument defines a timezone string, in the format described for the **TZ** environment variable. See the **environ(7)** man page.

A return value less than zero indicates a fatal error from a system call, most likely **malloc**.

The python bindings create a new zone handle and set reporting timezone for the timezone defined by *tz*.

pmUseZone Function

```
int pmUseZone(const int tz_handle)
Python:
int status = pmUseZone(int tz_handle)
```

In the **pmUseZone** function, *tz_handle* identifies a reporting timezone as previously established by a call to **pmNewZone** or **pmNewContextZone**, and this becomes the current reporting timezone. The current reporting timezone effects the timezone used by **pmCtime** and **pmLocaltime**).

A return value less than zero indicates the value of *tz_handle* is not legal.

The python bindings set the current reporting timezone defined by timezone *tz_handle*.

pmWhichZone Function

```
int pmWhichZone(char **tz)
Python:
"zone string" = pmWhichZone()
```

The **pmWhichZone** function returns the handle of the current timezone, as previously established by a call to **pmNewZone** or **pmNewContextZone**. If the call is successful (that is, there exists a current reporting timezone), a non-negative integer is returned and tz is set to point to a static buffer containing the timezone string itself. The current reporting timezone effects the timezone used by **pmCtime** and **pmLocaltime**.

A return value less than zero indicates there is no current reporting timezone.

The python bindings return the current reporting timezone.

14.8.6 PMAPI Metrics Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) metrics services.

pmFetch Function

```
int pmFetch(int numpmid, pmID pmidlist[], pmResult **result)
Python:
pmResult* pmresult = pmFetch(c_uint pmid[])
```

The most common PMAPI operation is likely to be calls to **pmFetch**, specifying a list of PMIDs (for example, as constructed by **pmLookupName**) through *pmidlist* and *numpmid*. The call to **pmFetch** is executed in the context of a source of metrics, instance profile, and collection time, previously established by calls to the functions described in Section 3.8.4, "*PMAPI Context Services*".

The principal result from **pmFetch** is returned as a tree structured *result*, described in the Section 3.5, "*Performance Metrics Values*".

If one value (for example, associated with a particular instance) for a requested metric is unavailable at the requested time, then there is no associated **pmValue** structure in the result. If there are no available values for a metric, then *numval* is zero and the associated **pmValue[]** instance is empty; *valfmt* is undefined in these circumstances, but *pmid* is correctly set to the PMID of the metric with no values.

If the source of the performance metrics is able to provide a reason why no values are available for a particular metric, this reason is encoded as a standard error code in the corresponding *numval*; see the **pmerr(1)** and **pmErrStr(3)** man pages. Since all error codes are negative, values for a requested metric are unavailable if *numval* is less than or equal to zero.

The argument definition and the result specifications have been constructed to ensure that for each PMID in the requested *pmidlist* there is exactly one **pmValueSet** in the result, and that the PMIDs appear in exactly the same sequence in both *pmidlist* and *result*. This makes the number and order of entries in *result* completely deterministic, and greatly simplifies the application programming logic after the call to **pmFetch**.

The result structure returned by **pmFetch** is dynamically allocated using one or more calls to **malloc** and specialized allocation strategies, and should be released when no longer required by calling **pmFreeResult**. Under no circumstances should **free** be called directly to release this space.

As common error conditions are encoded in the result data structure, only serious events (such as loss of connection to PMCD, **malloc** failure, and so on) would cause an error value to be returned by **pmFetch**. Otherwise, the value returned by the **pmFetch** function is zero.

In *Example 3.15. PMAPI Metrics Services*, the code fragment dumps the values (assumed to be stored in the *lval* element of the **pmValue** structure) of selected performance metrics once every 10 seconds:

Example 3.15. PMAPI Metrics Services

```
int       i, j, sts;
pmID      pmidlist[10];
pmResult  *result;
time_t    now;

/* set up PMAPI context, numpmid and pmidlist[] ... */
while ((sts = pmFetch(10, pmidlist, &result)) >= 0) {
    now = (time_t)result->timestamp.tv_sec;
    printf("\n@ %s", ctime(&now));
    for (i = 0; i < result->numpmid; i++) {
        printf("PMID: %s", pmIDStr(result->vset[i]->pmid));
        for (j = 0; j < result->vset[i]->numval; j++) {
            printf(" 0x%x", result->vset[i]->vlist[j].value.lval);
            putchar('\n');
        }
    }
    pmFreeResult(result);
    sleep(10);
}
```

Note: If a response is not received back from PMCD within 10 seconds, the **pmFetch** times out and returns **PM_ERR_TIMEOUT**. This is most likely to occur when the PMAPI client and PMCD are communicating over a slow network connection, but may also occur when one of the hosts is extremely busy. The time out period may be modified using the **PMCD_REQUEST_TIMEOUT** environment variable; see the **PCPIntro(1)** man page.

The python bindings fetch a pmResult corresponding to a *pmid* list, which is returned from **pmLookupName**. The returned *pmresult* is passed to **pmExtractValue**.

pmFreeResult Function

```
void pmFreeResult(pmResult *result)
Python:
pmFreeResult(pmResult* pmresult)
```

Release the storage previously allocated for a result by **pmFetch**.

THe python bindings free a *pmresult* previously allocated by **pmFetch**.

pmStore Function

```
int pmStore(const pmResult *request)
Python:
pmResult* pmresult = pmStore(pmResult* pmresult)
```

In some special cases it may be helpful to modify the current values of performance metrics in one or more underlying domains, for example to reset a counter to zero, or to modify a *metric*, which is a control variable within a Performance Metric Domain.

The **pmStore** function is a lightweight inverse of **pmFetch**. The caller must build the **pmResult** data structure (which could have been returned from an earlier **pmFetch** call) and then call **pmStore**. It is an error to pass a *request* to **pmStore** in which the **numval** field within any of the **pmValueSet** structure has a value less than one.

The current PMAPI context must be one with a host as the source of metrics, and the current value of the nominated metrics is changed. For example, **pmStore** cannot be used to make retrospective changes to information in a PCP archive log.

14.8.7 PMAPI Fetchgroup Services

The fetchgroup functions implement a registration-based mechanism to fetch groups of performance metrics, including automation for general unit, rate, type conversions and convenient instance and value encodings. They constitute a powerful and compact alternative to the classic Performance Metrics Application Programming Interface (PMAPI) sequence of separate lookup, check, fetch, iterate, extract, and convert functions.

A fetchgroup consists of a PMAPI context and a list of metrics that the application is interested in fetching. For each metric of interest, a conversion specification and a destination **pmAtomValue** pointer is given. Then, at each subsequent fetchgroup-fetch operation, all metrics are fetched, decoded/converted, and deposited in the desired field of the destination **pmAtomValues**. See *Example 3.18. pmAtomValue Structure* for more on that data type. Similarly, a per-metric-instance status value is optionally available for detailed diagnostics reflecting fetch/conversion.

The **pmfetchgroup(3)** man pages give detailed information on the C API; we only list some common cases here. The simplified Python binding to the same API is summarized below. One difference is that runtime errors in C are represented by status integers, but in Python are mapped to **pmErr** exceptions. Another is that supplying metric type codes are mandatory in the C API but optional in Python, since the latter language supports dynamic typing. Another difference is Python's wrapping of output metric values in callable "holder" objects. We demonstrate all of these below.

Fetchgroup setup

To create a fetchgroup and its private PMAPI context, the **pmCreateFetchGroup** function is used, with parameters similar to **pmNewContext** (see Section 3.8.4.1, "*pmNewContext Function*").

```
int sts;
pmFG fg;
sts = pmCreateFetchGroup(& fg, PM_CONTEXT_ARCHIVE, "./foo.meta");
assert(sts == 0);
Python
fg = pmapi.fetchgroup(c_api.PM_CONTEXT_ARCHIVE, './foo.meta')
```

If special PMAPI query, PMNS enumeration, or configuration upon the context is needed, the private context may be carefully accessed.

```
int ctx = pmGetFetchGroupContext(fg);
sts = pmUseContext(ctx);
assert(sts == 0);
sts = pmSetMode(...);
Python
ctx = fg.get_context()
ctx.pmSetMode(...)
```

A fetchgroup is born empty. It needs to be extended with metrics to read. Scalars are easy. We specify the metric name, an instance-domain instance if necessary, a unit-scaling and/or rate-conversion directive if desired, and a type code (see *Example 3.2. pmDesc Structure*). In C, the value destination is specified by pointer. In Python, a value-holder is returned.

```
static pmAtomValue ncpu, loadavg, idle;
sts = pmExtendFetchGroup_item(fg, "hinv.ncpu", NULL, NULL,
                              & ncpu, PM_TYPE_32, NULL);
assert (sts == 0);
sts = pmExtendFetchGroup_item(fg, "kernel.all.load", "5 minute", NULL,
                              & loadavg, PM_TYPE_DOUBLE, NULL);
assert (sts == 0);
sts = pmExtendFetchGroup_item(fg, "kernel.all.cpu.idle", NULL, "s/100s",
                              & idle, PM_TYPE_STRING, NULL);
assert (sts == 0);
Python
ncpu = fg.extend_item('hinv.cpu')
loadavg = fg.extend_item('kernel.all.load', instance='5 minute')
idle = fg.extend_item('kernel.all.cpu.idle, scale='s/100s')
```

Registering metrics with whole instance domains are also possible; these result in a vector of **pmAtomValue** instances, instance names and codes, and status codes, so the fetchgroup functions take more optional parameters. In Python, a value-holder-iterator object is returned.

```
enum { max_disks = 100 };
static unsigned num_disks;
static pmAtomValue disk_reads[max_disks];
static int disk_read_stss[max_disks];
static char *disk_names[max_disks];
sts = pmExtendFetchGroup_indom(fg, "disk.dm.read", NULL,
                               NULL, disk_names, disk_reads, PM_TYPE_32,
                               disk_read_stss, max_disks, & num_disks,
                               NULL);
Python
values = fg.extend_indom('disk.dm.read')
```

Registering interest in the future fetch-operation timestamp is also possible. In python, a datetime-holder object is returned.

```
struct timeval tv;
sts = pmExtendFetchGroup_timestamp(fg, & tv);
Python
tv = fg.extend_timestamp()
```

Fetchgroup operation

Now it's time for the program to process the metrics. In the C API, each metric value is put into status integers (if requested), and one field of the **pmAtomValue** union - whichever was requested with the **PM_TYPE_*** code. In the Python API, each metric value is accessed by calling the value-holder objects.

```
sts = pmFetchGroup(fg);
assert (sts == 0);
printf("%s", ctime(& tv.tv_sec));
printf("#cpus: %d, loadavg: %g, idle: %s\n", ncpu.l, loadavg.d, idle.cp);
for (i=0; i<num_disks; i++)
    if (disk_read_stss[i] == 0)
        printf("disk %s reads %d\n", disk_names[i], disk_reads[i].l);
Python
fg.fetch()
print(tv())
print("#cpus: %d, loadavg: %g, idle: %d\n" % (ncpu(), loadavg(), idle()))
for icode, iname, value in values():
    print('disk %s reads %d' % (iname, value()))
```

The program may fetch and process the values only once, or in a loop. The program need not - *must not* - modify or free any of the output values/pointers supplied by the fetchgroup functions.

Fetchgroup shutdown

Should the program wish to shut down a fetchgroup explicitly, thereby closing the private PMAPI context, there is a function for that.

```
sts = pmDestroyFetchGroup(fg);
Python
del fg # or nothing
```

14.8.8 PMAPI Record-Mode Services

The functions described in this section provide Performance Metrics Application Programming Interface (PMAPI) record-mode services. These services allow a monitor tool to establish connections to **pmlogger** co-processes, which they create and control for the purposes of recording live performance data from (possibly) multiple hosts. Since **pmlogger** records for one host only, these services can administer a group of loggers, and set up archive folios to track the logs. Tools like **pmafm** can subsequently use those folios to replay recorded data with the initiating tool. **pmchart** uses these concepts when providing its Record mode functionality.

pmRecordAddHost Function

```
int pmRecordAddHost(const char *host, int isdefault, pmRecordHost **rhp)
Python:
(int status, pmRecordHost* rhp) = pmRecordAddHost("host string", 1, "configure string
↪")
```

The **pmRecordAddHost** function adds hosts once **pmRecordSetup** has established a new recording session. The **pmRecordAddHost** function along with the **pmRecordSetup** and **pmRecordControl** functions are used to create a PCP archive.

pmRecordAddHost is called for each host that is to be included in the recording session. A new **pmRecordHost** structure is returned via *rhp*. It is assumed that PMCD is running on the host as this is how **pmlogger** retrieves the required performance metrics.

If this host is the default host for the recording session, *isdefault* is nonzero. This ensures that the corresponding archive appears first in the PCP archive *folio*. Hence the tools used to replay the archive *folio* make the correct determination of the archive associated with the default host. At most one host per recording session may be nominated as the default host.

The calling application writes the desired **pmlogger** configuration onto the stdio stream returned via the **f_config** field in the **pmRecordHost** structure.

pmRecordAddHost returns 0 on success and a value less than 0 suitable for decoding with **pmErrStr** on failure. The value **EINVAL** has the same interpretation as **errno** being set to **EINVAL**.

pmRecordControl Function

```
int pmRecordControl(pmRecordHost *rhp, int request, const char *options)
Python:
int status = pmRecordControl("host string", 1, "configure string")
```

Arguments may be optionally added to the command line that is used to launch **pmlogger** by calling the **pmRecordControl** function with a request of **PM_REC_SETARG**. The **pmRecordControl** along with the **pmRecordSetup** and **pmRecordAddHost** functions are used to create a PCP archive.

The argument is passed via *options* and one call to **pmRecordControl** is required for each distinct argument. An argument may be added for a particular **pmlogger** instance identified by *rhp*. If the *rhp* argument is NULL, the argument is added for all **pmlogger** instances that are launched in the current recording session.

Independent of any calls to **pmRecordControl** with a request of **PM_REC_SETARG**, each **pmlogger** instance is automatically launched with the following arguments: **-c, -h, -l, -x**, and the basename for the PCP archive log.

To commence the recording session, call **pmRecordControl** with a request of **PM_REC_ON**, and *rhp* must be NULL. This launches one **pmlogger** process for each host in the recording session and initializes the **fd_ipc, logfile, pid**, and **status** fields in the associated **pmRecordHost** structure(s).

To terminate a **pmlogger** instance identified by *rhp*, call **pmRecordControl** with a request of **PM_REC_OFF**. If the rhp argument to **pmRecordControl** is NULL, the termination request is broadcast to all **pmlogger** processes in the current recording session. An informative dialogue is generated directly by each **pmlogger** process.

To display the current status of the **pmlogger** instance identified by *rhp*, call **pmRecordControl** with a request of **PM_REC_STATUS**. If the *rhp* argument to **pmRecordControl** is NULL, the status request is broadcast to all **pmlogger** processes in the current recording session. The display is generated directly by each **pmlogger** process.

To detach a **pmlogger** instance identified by *rhp*, allow it to continue independent of the application that launched the recording session and call **pmRecordControl** with a request of **PM_REC_DETACH**. If the *rhp* argument to **pmRecordControl** is NULL, the detach request is broadcast to all **pmlogger** processes in the current recording session.

pmRecordControl returns 0 on success and a value less than 0 suitable for decoding with **pmErrStr** on failure. The value **EINVAL** has the same interpretation as **errno** being set to **EINVAL**.

pmRecordControl returns **PM_ERR_IPC** if the associated **pmlogger** process has already exited.

pmRecordSetup Function

```
FILE *pmRecordSetup(const char *folio, const char *creator, int replay)
Python:
int status = pmRecordSetup("folio string", "creator string", int replay)
```

The **pmRecordSetup** function along with the **pmRecordAddHost** and **pmRecordControl** functions may be used to create a PCP archive on the fly to support record-mode services for PMAPI client applications.

Each record mode session involves one or more PCP archive logs; each is created using a dedicated **pmlogger** process, with an overall Archive Folio format as understood by the **pmafm** command, to name and collect all of the archive logs associated with a single recording session.

The **pmRecordHost** structure is used to maintain state information between the creator of the recording session and the associated **pmlogger** process(es). The structure, shown in *Example 3.16. pmRecordHost Structure*, is defined as:

Example 3.16. pmRecordHost Structure

```
typedef struct {
    FILE   *f_config;    /* caller writes pmlogger configuration here */
    int    fd_ipc;       /* IPC channel to pmlogger */
    char   *logfile;     /* full pathname for pmlogger error logfile */
    pid_t  pid;          /* process id for pmlogger */
    int    status;       /* exit status, -1 if unknown */
} pmRecordHost;
```

In *Procedure 3.1. Creating a Recording Session*, the functions are used in combination to create a recording session.

Procedure 3.1. Creating a Recording Session

1. Call **pmRecordSetup** to establish a new recording session. A new Archive Folio is created using the name *folio*. If the *folio* file or directory already exists, or if it cannot be created, this is an error. The application that is creating the session is identified by creator (most often this would be the same as the global PMAPI application name, as returned **pmGetProgname**()). If the application knows how to create its own configuration file to replay the recorded session, replay should be nonzero. The **pmRecordSetup** function returns a stdio stream onto which the application writes the text of any required replay configuration file.
2. For each host that is to be included in the recording session, call **pmRecordAddHost**. A new **pmRecordHost** structure is returned via *rhp*. It is assumed that PMCD is running on the host as this is how **pmlogger** retrieves the required performance metrics. See Section 3.8.8.1, "*pmRecordAddHost Function*" for more information.
3. Optionally, add arguments to the command line that is used to launch **pmlogger** by calling **pmRecordControl** with a request of **PM_REC_SETARG**. The argument is passed via options and one call to **pmRecordControl** is required for each distinct argument. See Section 3.8.8.2, "*pmRecordControl Function*" for more information.
4. To commence the recording session, call **pmRecordControl** with a request of **PM_REC_ON**, and *rhp* must be NULL.
5. To terminate a **pmlogger** instance identified by *rhp*, call **pmRecordControl** with a request of **PM_REC_OFF**.
6. To display the current status of the **pmlogger** instance identified by *rhp*, call **pmRecordControl** with a request of **PM_REC_STATUS**.
7. To detach a **pmlogger** instance identified by *rhp*, allow it to continue independent of the application that launched the recording session, call **pmRecordControl** with a request of **PM_REC_DETACH**.

The calling application should not close any of the returned stdio streams; **pmRecordControl** performs this task when recording is commenced.

Once **pmlogger** has been started for a recording session, **pmlogger** assumes responsibility for any dialogue with the user in the event that the application that launched the recording session should exit, particularly without terminating

the recording session.

By default, information and dialogues from **pmlogger** is displayed using **pmconfirm**. This default is based on the assumption that most applications launching a recording session are GUI-based. In the event that **pmconfirm** fails to display the information (for example, because the **DISPLAY** environment variable is not set), **pmlogger** writes on its own stderr stream (not the stderr stream of the launching process). The output is assigned to the **xxxxxx.host.log** file. For convenience, the full pathname to this file is provided via the **logfile** field in the **pmRecordHost** structure.

If the *options* argument to **pmRecordControl** is not NULL, this string may be used to pass additional arguments to **pmconfirm** in those cases where a dialogue is to be displayed. One use of this capability is to provide a -geometry string to control the placement of the dialogue.

Premature termination of a launched **pmlogger** process may be determined using the **pmRecordHost** structure, by calling **select** on the **fd_ipc** field or polling the **status** field that will contain the termination status from **waitpid** if known, or -1.

These functions create a number of files in the same directory as the *folio* file named in the call to **pmRecordSetup**. In all cases, the *xxxxxx* component is the result of calling **mkstemp**.

- If replay is nonzero, *xxxxxx* is the creator's replay configuration file, else an empty control file, used to guarantee uniqueness.
- The *folio* file is the PCP Archive Folio, suitable for use with the **pmafm** command.
- The **xxxxxx.host.config** file is the **pmlogger** configuration for each host. If the same host is used in different calls to **pmRecordAddHost** within the same recording session, one of the letters 'a' through 'z' is appended to the *xxxxxx* part of all associated file names to ensure uniqueness.

  xxxxxx.host.log is stdout and stderr for the **pmlogger** instance for each host.
- The **xxxxxx.host.{0,meta,index}** files comprise a single PCP archive for each host.

pmRecordSetup may return NULL in the event of an error. Check **errno** for the real cause. The value **EINVAL** typically means that the order of calls to these functions is not correct; that is, there is an obvious state associated with the current recording session that is maintained across calls to the functions.

For example, calling **pmRecordControl** before calling **pmRecordAddHost** at least once, or calling **pmRecordAddHost** before calling **pmRecordSetup** would produce an **EINVAL** error.

14.8.9 PMAPI Archive-Specific Services

The functions described in this section provide archive-specific services.

pmGetArchiveLabel Function

```
int pmGetArchiveLabel(pmLogLabel *lp)
Python:
pmLogLabel loglabel = pmGetArchiveLabel()
```

Provided the current PMAPI context is associated with a set of PCP archive logs, the **pmGetArchiveLabel** function may be used to fetch the label record from the first archive in the set of archives. The structure returned through *lp* is as shown in *Example 3.17. pmLogLabel Structure*:

Example 3.17. pmLogLabel Structure

```
/*
 * Label Record at the start of every log file - as exported above the PMAPI ...
 */
```

(continues on next page)

(continued from previous page)

```
#define PM_TZ_MAXLEN    40
#define PM_LOG_MAXHOSTLEN   64
#define PM_LOG_MAGIC    0x50052600
#define PM_LOG_VERS01   0x1
#define PM_LOG_VERS02   0x2
#define PM_LOG_VOL_TI   -2      /* temporal index */
#define PM_LOG_VOL_META -1      /* meta data */
typedef struct {
    int             ll_magic;           /* PM_LOG_MAGIC | log format version no. */
    pid_t           ll_pid;             /* PID of logger */
    struct timeval ll_start;            /* start of this log */
    char            ll_hostname[PM_LOG_MAXHOSTLEN]; /* name of collection host */
    char            ll_tz[PM_TZ_MAXLEN];            /* $TZ at collection host */
} pmLogLabel;
```

The python bindings get the label record from the archive.

pmGetArchiveEnd Function

```
int pmGetArchiveEnd(struct timeval *tvp)
Python:
timeval tv = status = pmGetArchiveEnd()
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmGetArchiveEnd** finds the logical end of the last archive file in the set (after the last complete record in the archive), and returns the last recorded time stamp with *tvp*. This timestamp may be passed to **pmSetMode** to reliably position the context at the last valid log record, for example, in preparation for subsequent reading in reverse chronological order.

For archive logs that are not concurrently being written, the physical end of file and the logical end of file are co-incident. However, if an archive log is being written by **pmlogger** at the same time that an application is trying to read the archive, the logical end of file may be before the physical end of file due to write buffering that is not aligned with the logical record boundaries.

The python bindings get the last recorded timestamp from the archive.

pmGetInDomArchive Function

```
int pmGetInDomArchive(pmInDom indom, int **instlist, char ***namelist )
Python:
((instance1, instance2...) (name1, name2...)) pmGetInDom(pmDesc pmdesc)
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmGetInDomArchive** scans the metadata to generate the union of all instances for the instance domain *indom* that can be found in the set of archive logs, and returns through *instlist* the internal instance identifiers, and through *namelist* the full external identifiers.

This function is a specialized version of the more general PMAPI function **pmGetInDom**.

The function returns the number of instances found (a value less than zero indicates an error).

The resulting lists of instance identifiers (*instlist* and *namelist*), and the names that the elements of *namelist* point to, are allocated by **pmGetInDomArchive** with two calls to **malloc**, and it is the responsibility of the caller to use **free**(*instlist*) and **free**(*namelist*) to release the space when it is no longer required; see the **malloc(3)** and **free(3)** man pages.

When the result of **pmGetInDomArchive** is less than one, both *instlist* and *namelist* are undefined (no space is allocated; so calling **free** is a singularly bad idea).

The python bindings return a tuple of the instance IDs and names for the union of all instances for the instance domain *pmdesc* that can be found in the archive log.

pmLookupInDomArchive Function

```
int pmLookupInDomArchive(pmInDom indom, const char *name)
Python:
c_uint instid = pmLookupInDomArchive(pmDesc pmdesc, "Instance")
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmLookupInDomArchive** scans the metadata for the instance domain *indom*, locates the first instance with the external identification given by *name*, and returns the internal instance identifier.

This function is a specialized version of the more general PMAPI function **pmLookupInDom**.

The **pmLookupInDomArchive** function returns a positive instance identifier on success.

The python bindings return the instance id in *pmdesc* corresponding to *Instance*.

pmNameInDomArchive Function

```
int pmNameInDomArchive(pmInDom indom, int inst, char **name)
Python:
"instance id" = pmNameInDomArchive(pmDesc pmdesc, c_uint instid)
```

Provided the current PMAPI context is associated with a set of PCP archive logs, **pmNameInDomArchive** scans the metadata for the instance domain *indom*, locates the first instance with the internal instance identifier given by **inst**, and returns the full external instance identification through *name*. This function is a specialized version of the more general PMAPI function **pmNameInDom**.

The space for the value of *name* is allocated in **pmNameInDomArchive** with **malloc**, and it is the responsibility of the caller to free the space when it is no longer required; see the **malloc(3)** and **free(3)** man pages.

The python bindings return the text name of an instance corresponding to an instance domain *pmdesc* with instance identifier *instid*.

pmFetchArchive Function

```
int pmFetchArchive(pmResult **result)
Python:
pmResult* pmresult = pmFetchArchive()
```

This is a variant of **pmFetch** that may be used only when the current PMAPI context is associated with a set of PCP archive logs. The *result* is instantiated with all of the metrics (and instances) from the next archive record; consequently, there is no notion of a list of desired metrics, and the instance profile is ignored.

It is expected that **pmFetchArchive** would be used to create utilities that scan archive logs (for example, **pmdumplog** and **pmlogsummary**), and the more common access to the archives would be through the **pmFetch** interface.

14.8.10 PMAPI Time Control Services

The PMAPI provides a common framework for client applications to control time and to synchronize time with other applications. The user interface component of this service is fully described in the companion *Performance Co-Pilot User's and Administrator's Guide*. See also the **pmtime(1)** man page.

This service is most useful when processing sets of PCP archive logs, to control parameters such as the current archive position, update interval, replay rate, and timezone, but it can also be used in live mode to control a subset of these parameters. Applications such as **pmchart, pmgadgets, pmstat**, and **pmval** use the time control services to connect to an instance of the time control server process, **pmtime**, which provides a uniform graphical user interface to the time control services.

A full description of the PMAPI time control functions along with code examples can be found in man pages as listed in *Table 3.2. Time Control Functions in PMAPI*:

Table 3.2. Time Control Functions in PMAPI

Man Page	Synopsis of Time Control Function
pmCtime(3)	Formats the date and time for a reporting timezone.
pmLocaltime(3)	Converts the date and time for a reporting timezone.
pmParseTimeWindow(3)	Parses time window command line arguments.
pmTimeConnect(3)	Connects to a time control server via a command socket.
pmTimeDisconnect(3)	Closes the command socket to the time control server.
pmTimeGetPort(3)	Obtains the port name of the current time control server.
pmTimeRecv(3)	Blocks until the time control server sends a command message.
pmTimeSendAck(3)	Acknowledges completion of the step command.
pmTimeSendBounds(3)	Specifies beginning and end of archive time period.
pmTimeSendMode(3)	Requests time control server to change to a new VCR mode.
pmTimeSendPosition(3)	Requests time control server to change position or update intervals.
pmTimeSendTimezone(3)	Requests time control server to change timezones.
pmTimeShowDialog(3)	Changes the visibility of the time control dialogue.
pmTimeGetStatePixmap(3)	Returns array of pixmaps representing supplied time control state.

14.8.11 PMAPI Ancillary Support Services

The functions described in this section provide services that are complementary to, but not necessarily a part of, the distributed manipulation of performance metrics delivered by the PCP components.

pmGetConfig Function

```
char *pmGetConfig(const char *variable)
Python:
"env variable value = pmGetConfig("env variable")
```

The **pmGetConfig** function searches for a variable first in the environment and then, if one is not found, in the PCP configuration file and returns the string result. If a variable is not already in the environment, it is added with a call to the **setenv** function before returning.

The default location of the PCP configuration file is **/etc/pcp.conf**, but this location may be changed by setting **PCP_CONF** in the environment to a new location, as described in the **pcp.conf(5)** man page.

If the variable is not found in either the environment or the PCP configuration file (or the PCP configuration file is not found and **PCP_CONF** is not set in the environment), then a fatal error message is printed and the process will exit.

Although this sounds drastic, it is the only course of action available because the PCP configuration or installation is fatally flawed.

If this function returns, the returned value points to a string in the environment; and so although the function returns the same type as the **getenv** function (which should probably be a **const char ***), changing the content of the string is not recommended.

The python bindings return a value for environment variable *"env variable"* from environment or pcp config file.

pmErrStr Function

```
const char *pmErrStr(int code)
char *pmErrStr_r(int code, char *buf, int buflen);
Python:
"error string text" = pmErrStr(int error_code)
```

This function translates an error code into a text string, suitable for generating a diagnostic message. By convention within PCP, all error codes are negative. The small values are assumed to be negated versions of the platform error codes as defined in **errno.h**, and the strings returned are according to **strerror**. The large, negative error codes are PMAPI error conditions, and **pmErrStr** returns an appropriate PMAPI error string, as determined by *code*.

In the case of **pmErrStr**, the string value is held in a single static buffer, so concurrent calls may not produce the desired results. The **pmErrStr_r** function allows a buffer and length to be passed in, into which the message is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings return the error string corresponding to the *error code*.

pmExtractValue Function

```
int pmExtractValue(int valfmt, const pmValue *ival, int itype,
pmAtomValue *oval, int otype)
Python:
pmAtomValue atomval = pmExtractValue(int valfmt, const pmValue * ival,
              int itype,
              pmAtomValue *oval,
                int otype)
```

The **pmValue** structure is embedded within the **pmResult** structure, which is used to return one or more performance metrics; see the **pmFetch** man page.

All performance metric values may be encoded in a **pmAtomValue** union, defined in *Example 3.18. pmAtomValue Structure*:

Example 3.18. pmAtomValue Structure

```
/* Generic Union for Value-Type conversions */
typedef union {
    __int32_t  l;     /* 32-bit signed */
    __uint32_t ul;    /* 32-bit unsigned */
    __int64_t  ll;    /* 64-bit signed */
    __uint64_t ull;   /* 64-bit unsigned */
    float      f;     /* 32-bit floating point */
    double     d;     /* 64-bit floating point */
    char       *cp;   /* char ptr */
    void       *vp;   /* void ptr */
} pmAtomValue;
```

The **pmExtractValue** function provides a convenient mechanism for extracting values from the **pmValue** part of a **pmResult** structure, optionally converting the data type, and making the result available to the application programmer.

The *itype* argument defines the data type of the input value held in *ival* according to the storage format defined by *valfmt* (see the **pmFetch** man page). The *otype* argument defines the data type of the result to be placed in *oval*. The value for *itype* is typically extracted from a **pmDesc** structure, following a call to **pmLookupDesc** for a particular performance metric.

Table 3.3. PMAPI Type Conversion defines the various possibilities for the type conversion. The input type (*itype*) is shown vertically, and the output type (*otype*) horizontally. The following rules apply:

- Y means the conversion is always acceptable.
- N means conversion can never be performed (function returns **PM_ERR_CONV**).
- P means the conversion may lose accuracy (but no error status is returned).
- T means the result may be subject to high-order truncation (if this occurs the function returns **PM_ERR_TRUNC**).
- S means the conversion may be impossible due to the sign of the input value (if this occurs the function returns **PM_ERR_SIGN**).

If an error occurs, *oval* is set to zero (or NULL).

Note: Note that some of the conversions involving the **PM_TYPE_STRING** and **PM_TYPE_AGGREGATE** types are indeed possible, but are marked N; the rationale is that **pmExtractValue** should not attempt to duplicate functionality already available in the C library through **sscanf** and **sprintf**. No conversion involving the type **PM_TYPE_EVENT** is supported.

Table 3.3. PMAPI Type Conversion

TYPE	32	U32	64	U64	FLOAT	DBLE	STRING	AGGR	EVENT
32	Y	S	Y	S	P	P	N	N	N
U32	T	Y	Y	Y	P	P	N	N	N
64	T	T,S	Y	S	P	P	N	N	N
u64	T	T	T	Y	P	P	N	N	N
FLOAT	P,T	P,T,S	P,T	P,T,S	Y	Y	N	N	N
DBLE	P,T	P,T,S	P,T	P,T,S	P	Y	N	N	N
STRING	N	N	N	N	N	N	Y	N	N
AGGR	N	N	N	N	N	N	N	Y	N
EVENT	N	N	N	N	N	N	N	N	N

In the cases where multiple conversion errors could occur, the first encountered error is returned, and the order of checking is not defined.

If the output conversion is to one of the pointer types, such as *otype* **PM_TYPE_STRING** or **PM_TYPE_AGGREGATE**, then the value buffer is allocated by **pmExtractValue** using **malloc**, and it is the caller's responsibility to free the space when it is no longer required; see the **malloc(3)** and **free(3)** man pages.

Although this function appears rather complex, it has been constructed to assist the development of performance tools that convert values, whose type is known only through the **type** field in a **pmDesc** structure, into a canonical type for local processing.

The python bindings extract a value from a pmValue struct *ival* stored in format *valfmt* (see **pmFetch**), and convert its type from *itype* to *otype*.

pmConvScale Function

```
int
pmConvScale(int type, const pmAtomValue *ival, const pmUnits *iunit,
pmAtomValue *oval, pmUnits *ounit)
Python:
pmAtomValue atomval = pmConvScale(int itype, pmAtomValue value,
              pmDesc* pmdesc , int descidx, int otype)
```

Given a performance metric value pointed to by *ival*, multiply it by a scale factor and return the value in *oval*. The scaling takes place from the units defined by *iunit* into the units defined by *ounit*. Both input and output units must have the same dimensionality.

The performance metric type for both input and output values is determined by *type*, the value for which is typically extracted from a **pmDesc** structure, following a call to **pmLookupDesc** for a particular performance metric.

pmConvScale is most useful when values returned through **pmFetch** (and possibly extracted using **pmExtractValue**) need to be normalized into some canonical scale and units for the purposes of computation.

The python bindings convert a *value* pointed to by *pmdesc* entry *descidx* to a different scale *otype*.

pmUnitsStr Function

```
const char *pmUnitsStr(const pmUnits *pu)
char *pmUnitsStr_r(const pmUnits *pu, char *buf, int buflen)
Python:
"units string" = pmUnitsStr(pmUnits pmunits)
```

As an aid to labeling graphs and tables, or for error messages, **pmUnitsStr** takes a dimension and scale specification as per *pu*, and returns the corresponding text string.

pu is typically from a **pmDesc** structure, for example, as returned by **pmLookupDesc**.

If **pu* were **{1, -2, 0, PM_SPACE_MBYTE, PM_TIME_MSEC, 0}**, then the result string would be **Mbyte/sec^2**.

In the case of **pmUnitsStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmUnitsStr_r** function allows a buffer and length to be passed in, into which the units are stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmUnits struct *pmunits* to a readable string.

pmIDStr Function

```
const char *pmIDStr(pmID pmid)
char *pmIDStr_r(pmID pmid, char *buf, int buflen)
Python:
"ID string" = pmIDStr(int pmID)
```

For use in error and diagnostic messages, return a human readable version of the specified PMID, with each of the internal **domain, cluster**, and **item** subfields appearing as decimal numbers, separated by periods.

In the case of **pmIDStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmIDStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmID *pmid* to a readable string.

pmInDomStr Function

```
const char *pmInDomStr(pmInDom indom)
char *pmInDomStr_r(pmInDom indom, char *buf, int buflen)
Python:
"indom" = pmGetInDom(pmDesc pmdesc)
```

For use in error and diagnostic messages, return a human readable version of the specified instance domain identifier, with each of the internal **domain** and **serial** subfields appearing as decimal numbers, separated by periods.

In the case of **pmInDomStrr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmInDomStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate an instance domain ID pointed to by a pmDesc **pmdesc** to a readable string.

pmTypeStr Function

```
const char *pmTypeStr(int type)
char *pmTypeStr_r(int type, char *buf, int buflen)
Python:
"type" = pmTypeStr(int type)
```

Given a performance metric type, produce a terse ASCII equivalent, appropriate for use in error and diagnostic messages.

Examples are "32" (for **PM_TYPE_32**), "U64" (for **PM_TYPE_U64**), "AGGREGATE" (for **PM_TYPE_AGGREGATE**), and so on.

In the case of **pmTypeStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmTypeStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a performance metric type to a readable string. Constants are available for the types, e.g. c_api.PM_TYPE_FLOAT, by importing cpmapi.

pmAtomStr Function

```
const char *pmAtomStr(const pmAtomValue *avp, int type)
char *pmAtomStr_r(const pmAtomValue *avp, int typechar *buf, int buflen)
Python:
"value" = pmAtomStr(atom, type)
```

Given the **pmAtomValue** identified by *avp*, and a performance metric *type*, generate the corresponding metric value as a string, suitable for diagnostic or report output.

In the case of **pmAtomStr**, the string value is held in a single static buffer; so concurrent calls may not produce the desired results. The **pmAtomStr_r** function allows a buffer and length to be passed in, into which the identifier is stored; this variant uses no shared storage and can be used in a thread-safe manner.

The python bindings translate a pmAtomValue **atom** having performance metric **type** to a readable string. Constants are available for the types, e.g. c_api.PM_TYPE_U32, by importing cpmapi.

pmNumberStr Function

```
const char *pmNumberStr(double value)
char *pmNumberStr_r(double value, char *buf, int buflen)
```

The **pmNumberStr** function returns the address of a static 8-byte buffer that holds a null-byte terminated representation of value suitable for output with fixed-width fields.

The value is scaled using multipliers in powers of one thousand (the decimal kilo) and has a bias that provides greater precision for positive numbers as opposed to negative numbers. The format depends on the sign and magnitude of *value*.

pmPrintValue Function

```
void pmPrintValue(FILE *f, int valfmt, int type, const pmValue *val,
int minwidth)
Python:
pmPrintValue(FILE* file, pmResult pmresult, pmdesc, vset_index, vlist_index, min_
↪width)
```

The value of a single performance metric (as identified by *val*) is printed on the standard I/O stream identified by *f*. The value of the performance metric is interpreted according to the format of *val* as defined by *valfmt* (from a **pmValueSet** within a **pmResult**) and the generic description of the metric's type from a **pmDesc** structure, passed in through.

If the converted value is less than *minwidth* characters wide, it will have leading spaces to pad the output to a width of *minwidth* characters.

Example 3.19. Using pmPrintValue to Print Values illustrates using **pmPrintValue** to print the values from a **pmResult** structure returned via **pmFetch**:

Example 3.19. Using pmPrintValue to Print Values

```
int         numpmid, i, j, sts;
pmID        pmidlist[10];
pmDesc      desc[10];
pmResult    *result;

/* set up PMAPI context, numpmid and pmidlist[] ... */
/* get metric descriptors */
for (i = 0; i < numpmid; i++) {
    if ((sts = pmLookupDesc(pmidlist[i], &desc[i])) < 0) {
        printf("pmLookupDesc(pmid=%s): %s\n",
                        pmIDStr(pmidlist[i]), pmErrStr(sts));
        exit(1);
    }
}
if ((sts = pmFetch(numpmid, pmidlist, &result)) >= 0) {
    /* once per metric */
    for (i = 0; i < result->numpmid; i++) {
        printf("PMID: %s", pmIDStr(result->vset[i]->pmid));
        /* once per instance for this metric */
        for (j = 0; j < result->vset[i]->numval; j++) {
            printf(" [%d]", result->vset[i]->vlist[j].inst);
            pmPrintValue(stdout, result->vset[i]->valfmt,
                        desc[i].type,
                        &result->vset[i]->vlist[j],
                        8);
```

(continues on next page)

(continued from previous page)

```
        }
        putchar('\n');
    }
    pmFreeResult(result);
}
else
    printf("pmFetch: %s\n", pmErrStr(sts));
```

Print the value of a *pmresult* pointed to by *vset_index/vlist_index* and described by *pmdesc*. The format of a pmResult is described in *pmResult* The python bindings can use sys.__stdout__ as a value for file to display to stdout.

pmflush Function

```
int pmflush(void);
Python:
int status = pmflush()
```

The **pmflush** function causes the internal buffer which is shared with **pmprintf** to be either displayed in a window, printed on standard error, or flushed to a file and the internal buffer to be cleared.

The **PCP_STDERR** environment variable controls the output technique used by **pmflush**:

- If **PCP_STDERR** is unset, the text is written onto the stderr stream of the caller.
- If **PCP_STDERR** is set to the literal reserved word **DISPLAY**, then the text is displayed as a GUI dialogue using **pmconfirm**.

The **pmflush** function returns a value of zero on successful completion. A negative value is returned if an error was encountered, and this can be passed to **pmErrStr** to obtain the associated error message.

pmprintf Function

```
int pmprintf(const char *fmt, ... /*args*/);
Python:
pmprintf("fmt", ... /*args*/);
```

The **pmprintf** function appends the formatted message string to an internal buffer shared by the **pmprintf** and **pmflush** functions, without actually producing any output. The *fmt* argument is used to control the conversion, formatting, and printing of the variable length *args* list.

The **pmprintf** function uses the **mkstemp** function to securely create a **pcp**-prefixed temporary file in `${PCP_TMP_DIR}`. This temporary file is deleted when **pmflush** is called.

On successful completion, **pmprintf** returns the number of characters transmitted. A negative value is returned if an error was encountered, and this can be passed to **pmErrStr** to obtain the associated error message.

pmSortInstances Function

```
void pmSortInstances(pmResult *result)
Python:
pmSortInstances (pmResult* pmresult)
```

The **pmSortInstances** function may be used to guarantee that for each performance metric in the result from **pmFetch**, the instances are in ascending internal instance identifier sequence. This is useful when trying to compute rates from two consecutive **pmFetch** results, where the underlying instance domain or metric availability is not static.

pmParseInterval Function

```
int pmParseInterval(const char *string, struct timeval *rslt, char **errmsg)
Python:
(struct timeval, "error message") = pmParseInterval("time string")
```

The **pmParseInterval** function parses the argument string specifying an interval of time and fills in the **tv_sec** and **tv_usec** components of the **rslt** structure to represent that interval. The input string is most commonly the argument following a **-t** command line option to a PCP application, and the syntax is fully described in the **PCPIntro(1)** man page.

pmParseInterval returns 0 and *errmsg* is undefined if the parsing is successful. If the given string does not conform to the required syntax, the function returns -1 and a dynamically allocated error message string in *errmsg*.

The error message is terminated with a newline and includes the text of the input string along with an indicator of the position at which the error was detected as shown in the following example:

```
4minutes 30mumble
          ^ -- unexpected value
```

In the case of an error, the caller is responsible for calling **free** to release the space allocated for *errmsg*.

pmParseMetricSpec Function

```
int pmParseMetricSpec(const char *string, int isarch, char *source,
                      pmMetricSpec **rsltp, char **errmsg)
Python:
(pmMetricSpec metricspec, "error message") =
                      pmParseMetricSpec("metric specification", isarch, source)
```

The **pmParseMetricSpec** function accepts a *string* specifying the name of a PCP performance metric, and optionally the source (either a hostname, a set of PCP archive logs, or a local context) and instances for that metric. The syntax is described in the **PCPIntro(1)** man page.

If neither host nor archive component of the metric specification is provided, the **isarch** and **source** arguments are used to fill in the returned **pmMetricSpec** structure. In *Example 3.20. pmMetricSpec Structure*, the **pmMetricSpec** structure, which is returned via *rsltp*, represents the parsed string.

Example 3.20. pmMetricSpec Structure

```
typedef struct {
    int     isarch;      /* source type: 0 -> host, 1 -> archive, 2 -> local context␣
↪*/
    char    *source;     /* name of source host or archive */
    char    *metric;     /* name of metric */
```

(continues on next page)

(continued from previous page)

```
    int     ninst;      /* number of instances, 0 -> all */
    char    *inst[1];   /* array of instance names */
} pmMetricSpec;
```

The **pmParseMetricSpec** function returns 0 if the given string was successfully parsed. In this case, all the storage allocated by **pmParseMetricSpec** can be released by a single call to the **free** function by using the address returned from **pmMetricSpec** via *rsltp*. The convenience macro **pmFreeMetricSpec** is a thinly disguised wrapper for **free**.

The **pmParseMetricSpec** function returns 0 if the given string was successfully parsed. It returns **PM_ERR_GENERIC** and a dynamically allocated error message string in *errmsg* if the given string does not parse. In this situation, the error message string can be released with the **free** function.

In the case of an error, *rsltp* is undefined. In the case of success, *errmsg* is undefined. If *rsltp->ninst* is 0, then *rsltp->inst[0]* is undefined.

14.9 PMAPI Programming Issues and Examples

The following issues and examples are provided to enable you to create better custom performance monitoring tools.

The source code for a sample client (**pmclient**) using the PMAPI is shipped as part of the PCP package. See the **pmclient(1)** man page, and the source code, located in `${PCP_DEMOS_DIR}/pmclient`.

14.9.1 Symbolic Association between a Metric's Name and Value

A common problem in building specific performance tools is how to maintain the association between a performance metric's name, its access (instantiation) method, and the application program variable that contains the metric's value. Generally this results in code that is easily broken by bug fixes or changes in the underlying data structures. The PMAPI provides a uniform method for instantiating and accessing the values independent of the underlying implementation, although it does not solve the name-variable association problem. However, it does provide a framework within which a manageable solution may be developed.

Fundamentally, the goal is to be able to name a metric and reference the metric's value in a manner that is independent of the order of operations on other metrics; for example, to associate the **LOADAV** macro with the name **kernel.all.load**, and then be able to use **LOADAV** to get at the value of the corresponding metric.

The one-to-one association between the ordinal position of the metric names is input to **pmLookupName** and the PMIDs returned by this function, and the one-to-one association between the PMIDs input to **pmFetch** and the values returned by this function provide the basis for an automated solution.

The tool **pmgenmap** takes the specification of a list of metric names and symbolic tags, in the order they should be passed to **pmLookupName** and **pmFetch**. For example, **pmclient**:

```
          cat ${PCP_DEMOS_DIR}/pmclient/pmnsmap.spec
pmclient_init {
    hinv.ncpu                   NUMCPU
}

pmclient_sample {
    kernel.all.load    LOADAV
    kernel.percpu.cpu.user      CPU_USR
    kernel.percpu.cpu.sys       CPU_SYS
    mem.freemem                 FREEMEM
    disk.all.total              DKIOPS
}
```

This **pmgenmap** input produces the C code in *Example 3.21. C Code Produced by pmgenmap Input*. It is suitable for including with the **#include** statement:

Example 3.21. C Code Produced by pmgenmap Input

```
/*
 * Performance Metrics Name Space Map
 * Built by runme.sh from the file
 * pmnsmap.spec
 * on Thu Jan  9 14:13:49 EST 2014
 *
 * Do not edit this file!
 */

char *pmclient_init[] = {
#define NUMCPU 0
              "hinv.ncpu",

};


char *pmclient_sample[] = {
#define LOADAV 0
              "kernel.all.load",
#define CPU_USR        1
            "kernel.percpu.cpu.user",
#define CPU_SYS        2
              "kernel.percpu.cpu.sys",
#define FREEMEM        3
              "mem.freemem",
#define DKIOPS 4
              "disk.all.total",

};
```

14.9.2 Initializing New Metrics

Using the code generated by **pmgenmap**, you are now able to easily initialize the application's metric specifications as shown in *Example 3.22. Initializing Metric Specifications*:

Example 3.22. Initializing Metric Specifications

```
/* C code fragment from pmclient.c */
numpmid = sizeof(pmclient_sample) / sizeof(char *);
if ((pmidlist = (pmID *)malloc(numpmid * sizeof(pmidlist[0]))) == NULL) {...}
if ((sts = pmLookupName(numpmid, pmclient_sample, pmidlist)) < 0) {...}


# The equivalent python code would be
pmclient_sample = ("kernel.all.load", "kernel.percpu.cpu.user",
              "kernel.percpu.cpu.sys", "mem.freemem", "disk.all.total")
pmidlist = context.pmLookupName(pmclient_sample)
```

At this stage, **pmidlist** contains the PMID for the five metrics of interest.

14.9.3 Iterative Processing of Values

Assuming the tool is required to report values every *delta* seconds, use code similar to that in *Example 3.23. Iterative Processing*:

Example 3.23. Iterative Processing

```
/* censored C code fragment from pmclient.c */
while (samples == -1 || samples-- > 0) {
   if ((sts = pmFetch(numpmid, pmidlist, &crp)) < 0) { ... }
   for (i = 0; i < numpmid; i++)
      if ((sts = pmLookupDesc(pmidlist[i], &desclist[i])) < 0) { ... }
   ...
   pmExtractValue(crp->vset[FREEMEM]->valfmt, crp->vset[FREEMEM]->vlist,
                  desclist[FREEMEM].type, &tmp, PM_TYPE_FLOAT);
   pmConvScale(PM_TYPE_FLOAT, &tmp, &desclist[FREEMEM].units,
                    &atom, &mbyte_scale);
   ip->freemem = atom.f;
   ...
   __pmtimevalSleep(delta);
}


# The equivalent python code would be
FREEMEM = 3
desclist = context.pmLookupDescs(metric_names)
while (samples > 0):
    crp = context.pmFetch(metric_names)
    val = context.pmExtractValue(crp.contents.get_valfmt(FREEMEM),
                              crp.contents.get_vlist(FREEMEM, 0),
                              desclist[FREEMEM].contents.type,
                              c_api.PM_TYPE_FLOAT)
    atom = ctx.pmConvScale(c_api.PM_TYPE_FLOAT, val, desclist, FREEMEM,
                           c_api.PM_SPACE_MBYTE)
    (tvdelta, errmsg) = c_api.pmParseInterval(delta)
    c_api.pmtimevalSleep(delta)
```

14.9.4 Accommodating Program Evolution

The flexibility provided by the PMAPI and the **pmgenmap** utility is demonstrated by *Example 3.24. Adding a Metric*. Consider the requirement for reporting a third metric **mem.physmem**. This example shows how to add the line to the specification file:

Example 3.24. Adding a Metric

```
mem.freemem PHYSMEM
```

Then regenerate the **#include** file, and augment pmclient.c:

```
  pmExtractValue(crp->vset[PHYSMEM]->valfmt, crp->vset[PHYSMEM]->vlist,
                 desclist[PHYSMEM].type, &tmp, PM_TYPE_FLOAT);
  pmConvScale(PM_TYPE_FLOAT, &tmp, &desclist[PHYSMEM].units,
                   &atom, &mbyte_scale);


# The equivalent python code would be:
```

(continues on next page)

(continued from previous page)

```
val = context.pmExtractValue(crp.contents.get_valfmt(PHYSMEM),
                        crp.contents.get_vlist(PHYSMEM, 0),
                        desclist[PHYSMEM].contents.type,
                        c_api.PM_TYPE_FLOAT);
```

14.9.5 Handling PMAPI Errors

In *Example 3.25. PMAPI Error Handling*, the simple but complete PMAPI application demonstrates the recommended style for handling PMAPI error conditions. The python bindings use the exception mechanism to raise an exception in error cases. The python client can handle this condition by catching the **pmErr** exception. For simplicity, no command line argument processing is shown here - in practice most tools use the **pmGetOptions** helper interface to assist with initial context creation and setup.

Example 3.25. PMAPI Error Handling

```
#include <pcp/pmapi.h>

int
main(int argc, char* argv[])
{
    int                 sts = 0;
    char                *host = "local:";
    char                *metric = "mem.freemem";
    pmID                pmid;
    pmDesc              desc;
    pmResult            *result;

    sts = pmNewContext(PM_CONTEXT_HOST, host);
    if (sts < 0) {
        fprintf(stderr, "Error connecting to pmcd on %s: %s\n",
                host, pmErrStr(sts));
        exit(1);
    }
    sts = pmLookupName(1, &metric, &pmid);
    if (sts < 0) {
        fprintf(stderr, "Error looking up %s: %s\n", metric,
                pmErrStr(sts));
        exit(1);
    }
    sts = pmLookupDesc(pmid, &desc);
    if (sts < 0) {
        fprintf(stderr, "Error getting descriptor for %s:%s: %s\n",
                host, metric, pmErrStr(sts));
        exit(1);
    }
    sts = pmFetch(1, &pmid, &result);
    if (sts < 0) {
        fprintf(stderr, "Error fetching %s:%s: %s\n", host, metric,
                pmErrStr(sts));
        exit(1);
    }
    sts = result->vset[0]->numval;
    if (sts < 0) {
        fprintf(stderr, "Error fetching %s:%s: %s\n", host, metric,
                pmErrStr(sts));
```

(continues on next page)

(continued from previous page)

```
        exit(1);
    }
    fprintf(stdout, "%s:%s = ", host, metric);
    if (sts == 0)
        puts("(no value)");
    else {
        pmValueSet      *vsp = result->vset[0];
        pmPrintValue(stdout, vsp->valfmt, desc.type,
                             &vsp->vlist[0], 5);
        printf(" %s\n", pmUnitsStr(&desc.units));
    }
    return 0;
}
```

The equivalent python code would be:

```
import sys
import traceback
from pcp import pmapi
from cpmapi import PM_TYPE_U32

try:
    context = pmapi.pmContext()
    pmid = context.pmLookupName("mem.freemem")
    desc = context.pmLookupDescs(pmid)
    result = context.pmFetch(pmid)
    freemem = context.pmExtractValue(result.contents.get_valfmt(0),
                                     result.contents.get_vlist(0, 0),
                                     desc[0].contents.type,
                                     PM_TYPE_U32)
    print "freemem is " + str(int(freemem.ul))

except pmapi.pmErr, error:
    print "%s: %s" % (sys.argv[0], error.message())
except Exception, error:
    sys.stderr.write(str(error) + "\n")
    sys.stderr.write(traceback.format_exc() + "\n")
```

14.9.6 Compiling and Linking PMAPI Applications

Typical PMAPI applications require the following line to include the function prototype and data structure definitions used by the PMAPI.

```
#include <pcp/pmapi.h>
```

Some applications may also require these header files: **<pcp/libpcp.h>** and **<pcp/pmda.h>**.

The run-time environment of the PMAPI is mostly found in the **libpcp** library; so to link a generic PMAPI application requires something akin to the following command:

```
cc mycode.c -lpcp
```

CHAPTER

FIFTEEN

INSTRUMENTING APPLICATIONS

Contents

- *Instrumenting Applications*
  - *Application and Performance Co-Pilot Relationship*
  - *Performance Instrumentation and Sampling*
  - *MMV PMDA Design*
  - *Memory Mapped Values API*
    - *Starting and Stopping Instrumentation*
    - *Getting a Handle on Mapped Values*
    - *Updating Mapped Values*
    - *Elapsed Time Measures*
  - *Performance Instrumentation and Tracing*
  - *Trace PMDA Design*
    - *Application Interaction*
    - *Sampling Techniques*
      - *Simple Periodic Sampling*
      - *Rolling-Window Periodic Sampling*
      - *Rolling-Window Periodic Sampling Example*
    - *Configuring the Trace PMDA*
  - *Trace API*
    - *Transactions*
    - *Point Tracing*
    - *Observations and Counters*
    - *Configuring the Trace Library*

This chapter provides an introduction to ways of instrumenting applications using PCP.

The first section covers the use of the Memory Mapped Value (MMV) Performance Metrics Domain Agent (PMDA) to generate customized metrics from an application. This provides a robust, extremely efficient mechanism for transfer-

ring custom instrumentation into the PCP infrastructure. It has been successfully deployed in production environments for many years, has proven immensely valuable in these situations, and can be used to instrument applications written in a number of programming languages.

The Memory Mapped Value library and PMDA is supported on every PCP platform, and is enabled by default.

Note: A particularly expansive Java API is available from the separate Parfait project. It supports both the existing JVM instrumentation, and custom application metric extensions.

The chapter also includes information on how to use the MMV library (**libpcp_mmv**) for instrumenting an application. The example programs are installed in `${PCP_DEMOS_DIR}/mmv`.

The second section covers the design of the Trace PMDA, in an effort to explain how to configure the agent optimally for a particular problem domain. This information supplements the functional coverage which the man pages provide to both the agent and the library interfaces.

This part of the chapter also includes information on how to use the Trace PMDA and its associated library (**libpcp_trace**) for instrumenting applications. The example programs are installed in `${PCP_DEMOS_DIR}/trace`.

Warning: The current PCP trace library is a relatively heavy-weight solution, issuing multiple system calls per trace point, runs over a TCP/IP socket even locally and performs no event batching. As such it is not appropriate for production application instrumentation at this stage.

A revised application tracing library and PMDA are planned which will be light-weight, suitable for production system tracing, and support event metrics and other advances in end-to-end distributed application tracing.

The application instrumentation libraries are designed to encourage application developers to embed calls in their code that enable application performance data to be exported. When combined with system-level performance data, this feature allows total performance and resource demands of an application to be correlated with application activity.

For example, developers can provide the following application performance metrics:

- Computation state (especially for codes with major shifts in resource demands between phases of their execution)
- Problem size and parameters, that is, degree of parallelism throughput in terms of sub-problems solved, iteration count, transactions, data sets inspected, and so on
- Service time by operation type

15.1 Application and Performance Co-Pilot Relationship

The relationship between an application, the **pcp_mmv** and **pcp_trace** instrumentation libraries, the MMV and Trace PMDAs, and the rest of the PCP infrastructure is shown in *Figure 4.1. Application and PCP Relationship*:

Fig. 1: Figure 4.1. Application and PCP Relationship

Once the application performance metrics are exported into the PCP framework, all of the PCP tools may be leveraged to provide performance monitoring and management, including:

- Two- and three-dimensional visualization of resource demands and performance, showing concurrent system activity and application activity.

- Transport of performance data over the network for distributed performance management.
- Archive logging for historical records of performance, most useful for problem diagnosis, postmortem analysis, performance regression testing, capacity planning, and benchmarking.
- Automated alarms when bad performance is observed. These apply both in real-time or when scanning archives of historical application performance.

15.2 Performance Instrumentation and Sampling

The **pcp_mmv** library provides function calls to assist with extracting important performance metrics from a program into a shared, in-memory location such that the MMV PMDA can examine and serve that information on behalf of PCP client tool requests. The **pcp_mmv** library is described in the **mmv_stats_init(3)**, **mmv_lookup_value_desc(3)**, **mmv_inc_value(3)** man pages. Additionally, the format of the shared memory mappings is described in detail in **mmv(5)**.

15.3 MMV PMDA Design

An application instrumented with memory mapped values directly updates the memory that backs the metric values it exports. The MMV PMDA reads those values directly, from the **same** memory that the application is updating, when current values are sampled on behalf of PMAPI client tools. This relationship, and a simplified MMV API, are shown in *Figure 4.2. Memory Mapped Page Sharing*.

Fig. 2: Figure 4.2. Memory Mapped Page Sharing

It is worth noting that once the metrics of an application have been registered via the **pcp_mmv** library initialisation API, subsequent interactions with the library are not intrusive to the instrumented application. At the points where values are updated, the only cost involved is the memory mapping update, which is a single memory store operation. There is no need to explicitly transfer control to the MMV PMDA, nor allocate memory, nor make system or library calls. The PMDA will only sample the values at times driven by PMAPI client tools, and this places no overhead on the instrumented application.

15.4 Memory Mapped Values API

The **libpcp_mmv** Application Programming Interface (API) can be called from C, C++, Perl and Python (a separate project, Parfait, services the needs of Java applications). Each language has access to the complete set of functionality offered by **libpcp_mmv**. In most cases, the calling conventions differ only slightly between languages - in the case of Java and Parfait, they differ significantly however.

15.4.1 Starting and Stopping Instrumentation

Instrumentation is begun with an initial call to **mmv_stats_init**, and ended with a call to **mmv_stats_stop**. These calls manipulate global state shared by the library and application. These are the only calls requiring synchonization and a single call to each is typically performed early and late in the life of the application (although they can be used to reset the library state as well, at any time). As such, the choice of synchonization primitive is left to the application, and none is currently performed by the library.

```
void *mmv_stats_init(const char *name, int cluster, mmv_stats_flags_t flags,
                     const mmv_metric_t *stats, int nstats,
                     const mmv_indom_t *indoms, int nindoms)
```

The *name* should be a simple symbolic name identifying the application. It is usually used as the first application-specific part of the exported metric names, as seen from the MMV PMDA. This behavior can be overriden using the *flags* parameter, with the MMV_FLAG_NOPREFIX flag. In the example below, full metric names such as **mmv.acme.products.count** will be created by the MMV PMDA. With the MMV_FLAG_NOPREFIX flag set, that would instead become **mmv.products.count**. It is recommended to not disable the prefix - doing so requires the applications to ensure naming conflicts do not arise in the MMV PMDA metric names.

The *cluster* identifier is used by the MMV PMDA to further distinguish different applications, and is directly used for the MMV PMDA PMID cluster field described in *Example 2.3. __pmID_int Structure*, for all MMV PMDA metrics.

All remaining parameters to **mmv_stats_init** define the metrics and instance domains that exist within the application. These are somewhat analogous to the final parameters of **pmdaInit(3)**, and are best explained using *Example 4.1. Memory Mapped Value Instance Structures* and *Example 4.2. Memory Mapped Value Metrics Structures*. As mentioned earlier, the full source code for this example instrumented application can be found in `${PCP_DEMOS_DIR}/mmv`.

Example 4.1. Memory Mapped Value Instance Structures

```
#include <pcp/pmapi.h>
#include <pcp/mmv_stats.h>

static mmv_instances_t products[] = {
    {   .internal = 0, .external = "Anvils" },
    {   .internal = 1, .external = "Rockets" },
    {   .internal = 2, .external = "Giant_Rubber_Bands" },
};
#define ACME_PRODUCTS_INDOM 61
#define ACME_PRODUCTS_COUNT (sizeof(products)/sizeof(products[0]))

static mmv_indom_t indoms[] = {
    {   .serial = ACME_PRODUCTS_INDOM,
        .count = ACME_PRODUCTS_COUNT,
        .instances = products,
        .shorttext = "Acme products",
        .helptext = "Most popular products produced by the Acme Corporation",
    },
};
```

The above data structures initialize an instance domain of the set of products produced in a factory by the fictional "Acme Corporation". These structures are directly comparable to several concepts we have seen already (and for good reason - the MMV PMDA must interpret the applications intentions and properly export instances on its behalf):

- mmv_instances_t maps to pmdaInstid, as in *Example 2.7. pmdaInstid Structure*
- mmv_indom_t maps to pmdaIndom, as in *Example 2.8. pmdaIndom Structure* - the major difference is the addition of oneline and long help text, the purpose of which should be self-explanatory at this stage.

- *serial* numbers, as in *Example 2.9. __pmInDom_int Structure*

Next, we shall create three metrics, all of which use this instance domain. These are the **mmv.acme.products** metrics, and they reflect the rates at which products are built by the machines in the factory, how long these builds take for each product, and how long each product type spends queued (while waiting for factory capacity to become available).

Example 4.2. Memory Mapped Value Metrics Structures

```
static mmv_metric_t metrics[] = {
    {   .name = "products.count",
        .item = 7,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
        .dimension = MMV_UNITS(0,0,1,0,0,PM_COUNT_ONE),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Acme factory product throughput",
        .helptext =
"Monotonic increasing counter of products produced in the Acme Corporation\n"
"factory since starting the Acme production application.  Quality guaranteed.",
    },
    {   .name = "products.time",
        .item = 8,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
        .dimension = MMV_UNITS(0,1,0,0,PM_TIME_USEC,0),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Machine time spent producing Acme products",
        .helptext =
"Machine time spent producing Acme Corporation products.  Does not include\n"
"time in queues waiting for production machinery.",
    },
    {   .name = "products.queuetime",
        .item = 10,
        .type = MMV_TYPE_U64,
        .semantics = MMV_SEM_COUNTER,
        .dimension = MMV_UNITS(0,1,0,0,PM_TIME_USEC,0),
        .indom = ACME_PRODUCTS_INDOM,
        .shorttext = "Queued time while producing Acme products",
        .helptext =
"Time spent in the queue waiting to build Acme Corporation products,\n"
"while some other Acme product was being built instead of this one.",
    },
};
#define INDOM_COUNT (sizeof(indoms)/sizeof(indoms[0]))
#define METRIC_COUNT (sizeof(metrics)/sizeof(metrics[0]))
```

As was the case with the "products" instance domain before, these metric-defining data structures are directly comparable to PMDA data structures described earlier:

- mmv_metric_t maps to a pmDesc structure, as in *Example 3.2. pmDesc Structure*
- MMV_TYPE, MMV_SEM, and MMV_UNITS map to PMAPI constructs for type, semantics, dimensionality and scale, as in *Example 3.3. pmUnits and pmDesc Structures*
- *item* number, as in *Example 2.3. __pmID_int Structure*

For the most part, all types and macros map directly to their core PCP counterparts, which the MMV PMDA will use when exporting the metrics. One important exception is the introduction of the metric type MMV_TYPE_ELAPSED, which is discussed further in Section 4.4.4, "*Elapsed Time Measures*".

The compound metric types - aggregate and event type metrics - are not supported by the MMV format.

15.4.2 Getting a Handle on Mapped Values

Once metrics (and the instance domains they use) have been registered, the memory mapped file has been created and is ready for use. In order to be able to update the individual metric values, however, we must find get a handle to the value. This is done using the **mmv_lookup_value_desc** function, as shown in *Example 4.3. Memory Mapped Value Handles*.

Example 4.3. Memory Mapped Value Handles

```
#define ACME_CLUSTER 321        /* PMID cluster identifier */

int
main(int argc, char * argv[])
{
    void *base;
    pmAtomValue *count[ACME_PRODUCTS_COUNT];
    pmAtomValue *machine[ACME_PRODUCTS_COUNT];
    pmAtomValue *inqueue[ACME_PRODUCTS_COUNT];
    unsigned int working;
    unsigned int product;
    unsigned int i;

    base = mmv_stats_init("acme", ACME_CLUSTER, 0,
                          metrics, METRIC_COUNT, indoms, INDOM_COUNT);
    if (!base) {
        perror("mmv_stats_init");
        return 1;
    }

    for (i = 0; i < ACME_PRODUCTS_COUNT; i++) {
        count[i] = mmv_lookup_value_desc(base,
                        "products.count", products[i].external);
        machine[i] = mmv_lookup_value_desc(base,
                        "products.time", products[i].external);
        inqueue[i] = mmv_lookup_value_desc(base,
                        "products.queuetime", products[i].external);
    }
```

Space in the mapping file for every value is set aside at initialization time (by the **mmv_stats_init** function) - that is, space for each and every metric, and each value (instance) of each metric when an instance domain is used. To find the handle to the space set aside for one individual value requires the tuple of base memory address of the mapping, metric name, and instance name. In the case of metrics with no instance domain, the final instance name parameter should be either NULL or the empty string.

15.4.3 Updating Mapped Values

At this stage we have individual handles (pointers) to each instrumentation point, we can now start modifying these values and observing changes through the PCP infrastructure. Notice that each handle is simply the canonical **pmAtomValue** pointer, as defined in *Example 3.18. pmAtomValue Structure*, which is a union providing sufficient space to hold any single value.

This pointer can be either manipulated directly, or using helper functions provided by the **pcp_mmv** API, such as the **mmv_stats_inc** and **mmv_stats_set** functions.

Example 4.4. Memory Mapped Value Updates

```
while (1) {
    /* choose a random number between 0-N -> product */
    product = rand() % ACME_PRODUCTS_COUNT;

    /* assign a time spent "working" on this product */
    working = rand() % 50000;

    /* pretend to "work" so process doesn't burn CPU */
    usleep(working);

    /* update the memory mapped values for this one: */
    /* one more product produced and work time spent */
    mmv_inc_value(base, machine[product], working); /* API */
    count[product]->ull += 1;     /* or direct mmap update */

    /* all other products are "queued" for this time */
    for (i = 0; i < ACME_PRODUCTS_COUNT; i++)
        if (i != product)
            mmv_inc_value(base, inqueue[i], working);
}
```

At this stage, it will be informative to compile and run the complete example program, which can be found in `${PCP_DEMOS_DIR}/mmv/acme.c`. There is an associated **Makefile** to build it, in the same directory. Running the **acme** binary creates the instrumentation shown in *Example 4.5. Memory Mapped Value Reports*, with live values letting us explore simple queueing effects in products being created on the ACME factory floor.

Example 4.5. Memory Mapped Value Reports

```
pminfo -m mmv.acme
mmv.acme.products.queuetime PMID: 70.321.10
mmv.acme.products.time PMID: 70.321.8
mmv.acme.products.count PMID: 70.321.7

pmval -f2 -s3 mmv.acme.products.time
metric:    mmv.acme.products.time
host:      localhost
semantics: cumulative counter (converting to rate)
units:     microsec (converting to time utilization)
samples:   3
interval:  1.00 sec

               Anvils              Rockets    Giant_Rubber_Bands
                 0.37                 0.12                  0.50
                 0.35                 0.25                  0.38
                 0.57                 0.20                  0.23
```

Experimentation with the algorithm from *Example 4.4. Memory Mapped Value Updates* is encouraged. In particular, observe the effects of rate conversion (counter metric type) of a metric with units of "time" (PM_TIME_*). The reported values are calculated over a sampling interval, which also has units of "time", forming a utilization. This is extremely valuable performance analysis currency - comparable metrics would include processor utilization, disk spindle utilization, and so forth.

15.4.4 Elapsed Time Measures

One problem with the instrumentation model embodied by the **pcp_mmv** library is providing timing information for long-running operations. For instrumenting long-running operations, like uploading downloading a file, the overall operation may be broken into smaller, discrete units of work which can be easily instrumented in terms of operations and throughput measures. In other cases, there are no divisible units for long-running operations (for example a black-box library call) and instrumenting these operations presents a challenge. Sometimes the best that can be done is adding the instrumentation point at the completion of the operation, and simply accept the "bursty" nature of this approach. In these problematic cases, the work completed in one sampling-interval may have begun several intervals before, from the point of view of the monitoring tool, which can lead to misleading results.

One technique that is available to combat this is through use of the MMV_TYPE_ELAPSED metric type, which provides the concept of a "timed section" of code. This mechanism stores the start time of an operation along with the mapped metric value (an "elapsed time" counter), via the **mmv_stats_interval_start** instrumentation function. Then, with help from the MMV PMDA which recognizes this type, the act of sampling the metric value causes an **interim** timestamp to be taken (by the MMV PMDA, not the application) and **combined** with the initial timestamp to form a more accurate reflection of time spent within the timed section, which effectively smooths out the bursty nature of the instrumentation.

The completion of each timed section of code is marked by a call to **mmv_stats_interval_end** which signifies to the MMV PMDA that the operation is not active, and no extra "in-progress" time should be applied to the exported value. At that time, the elapsed time for the entire operation is calculated and accounted toward metrics value.

15.5 Performance Instrumentation and Tracing

The **pcp_trace** library provides function calls for identifying sections of a program as transactions or events for examination by the trace PMDA, a user command called **pmdatrace**. The **pcp_trace** library is described in the **pmdatrace(3)** man page.

The monitoring of transactions using the Performance Co-Pilot (PCP) infrastructure begins with a **pmtracebegin** call. Time is recorded from there to the corresponding **pmtraceend** call (with matching tag identifier). A transaction in progress can be cancelled by calling **pmtraceabort**.

A second form of program instrumentation is available with the **pmtracepoint** function. This is a simpler form of monitoring that exports only the number of times a particular point in a program is passed. The **pmtraceobs** and **pmtracecount** functions have similar semantics, but the former allows an arbitrary numeric value to be passed to the trace PMDA.

The **pmdatrace** command is a PMDA that exports transaction performance metrics from application processes using the **pcp_trace** library; see the **pmdatrace(1)** man page for details.

15.6 Trace PMDA Design

Trace PMDA design covers application interaction, sampling techniques, and configuring the trace PMDA.

15.6.1 Application Interaction

Figure 4.3. Trace PMDA Overview describes the general state maintained within the trace PMDA.

Fig. 3: Figure 4.3. Trace PMDA Overview

Applications that are linked with the **libpcp_trace** library make calls through the trace Application Programming Interface (API). These calls result in interprocess communication of trace data between the application and the trace PMDA. This data consists of an identification tag and the performance data associated with that particular tag. The trace PMDA aggregates the incoming information and periodically updates the exported summary information to describe activity in the recent past.

As each protocol data unit (PDU) is received, its data is stored in the current working buffer. At the same time, the global counter associated with the particular tag contained within the PDU is incremented. The working buffer contains all performance data that has arrived since the previous time interval elapsed. For additional information about the working buffer, see Section 4.6.2.2, "*Rolling-Window Periodic Sampling*".

15.6.2 Sampling Techniques

The trace PMDA employs a rolling-window periodic sampling technique. The arrival time of the data at the trace PMDA in conjunction with the length of the sampling period being maintained by the PMDA determines the recency of the data exported by the PMDA. Through the use of rolling-window sampling, the trace PMDA is able to present a more accurate representation of the available trace data at any given time than it could through use of simple periodic sampling.

The rolling-window sampling technique affects the metrics in *Example 4.6. Rolling-Window Sampling Technique*:

Example 4.6. Rolling-Window Sampling Technique

```
trace.observe.rate
trace.counter.rate
trace.point.rate
trace.transact.ave_time
trace.transact.max_time
trace.transact.min_time
trace.transact.rate
```

The remaining metrics are either global counters, control metrics, or the last seen observation value. Section 4.7, "*Trace API*", documents in more detail all metrics exported by the trace PMDA.

Simple Periodic Sampling

The simple periodic sampling technique uses a single historical buffer to store the history of events that have occurred over the sampling interval. As events occur, they are recorded in the working buffer. At the end of each sampling interval, the working buffer (which at that time holds the historical data for the sampling interval just finished) is copied into the historical buffer, and the working buffer is cleared. It is ready to hold new events from the sampling interval now starting.

Rolling-Window Periodic Sampling

In contrast to simple periodic sampling with its single historical buffer, the rolling-window periodic sampling technique maintains a number of separate buffers. One buffer is marked as the current working buffer, and the remainder of the buffers hold historical data. As each event occurs, the current working buffer is updated to reflect it.

At a specified interval, the current working buffer and the accumulated data that it holds is moved into the set of historical buffers, and a new working buffer is used. The specified interval is a function of the number of historical buffers maintained.

The primary advantage of the rolling-window sampling technique is seen at the point where data is actually exported. At this point, the data has a higher probability of reflecting a more recent sampling period than the data exported using simple periodic sampling.

The data collected over each sample duration and exported using the rolling-window sampling technique provides a more up-to-date representation of the activity during the most recently completed sample duration than simple periodic sampling as shown in *Figure 4.4. Sample Duration Comparison*.

Fig. 4: Figure 4.4. Sample Duration Comparison

The trace PMDA allows the length of the sample duration to be configured, as well as the number of historical buffers that are maintained. The rolling-window approach is implemented in the trace PMDA as a ring buffer (see *Figure 4.3. Trace PMDA Overview*).

When the current working buffer is moved into the set of historical buffers, the least recent historical buffer is cleared of data and becomes the new working buffer.

Rolling-Window Periodic Sampling Example

Consider the scenario where you want to know the rate of transactions over the last 10 seconds. You set the sampling rate for the trace PMDA to 10 seconds and fetch the metric **trace.transact.rate**. So if in the last 10 seconds, 8 transactions took place, the transaction rate would be 8/10 or 0.8 transactions per second.

The trace PMDA does not actually do this. It instead does its calculations automatically at a subinterval of the sampling interval. Reconsider the 10-second scenario. It has a calculation subinterval of 2 seconds as shown in *Figure 4.5. Sampling Intervals*.

Fig. 5: Figure 4.5. Sampling Intervals

If at 13.5 seconds, you request the transaction rate, you receive a value of 0.7 transactions per second. In actual fact, the transaction rate was 0.8, but the trace PMDA did its calculations on the sampling interval from 2 seconds to 12 seconds, and not from 3.5 seconds to 13.5 seconds. For efficiency, the trace PMDA calculates the metrics on the last 10 seconds every 2 seconds. As a result, the PMDA is not driven each time a fetch request is received to do a calculation.

15.6.3 Configuring the Trace PMDA

The trace PMDA is configurable primarily through command-line options. The list of command-line options in *Table 4.1. Selected Command-Line Options* is not exhaustive, but it identifies those options which are particularly relevant to tuning the manner in which performance data is collected.

Table 4.1. Selected Command-Line Options

Option	Description
Access controls	The trace PMDA offers host-based access control. This control allows and disallows connections from instrumented applications running on specified hosts or groups of hosts. Limits to the number of connections allowed from individual hosts can also be mandated.
Sample duration	The interval over which metrics are to be maintained before being discarded is called the sample duration.
Number of historical buffers	The data maintained for the sample duration is held in a number of internal buffers within the trace PMDA. These are referred to as historical buffers. This number is configurable so that the rolling window effect can be tuned within the sample duration.
Counter and observation metric units	Since the data being exported by the **trace.observe.value** and **trace.counter.count** metrics are user-defined, the trace PMDA by default exports these metrics with a type of "none." A framework is provided that allows the user to make the type more specific (for example, bytes per second) and allows the exported values to be plotted along with other performance metrics of similar units by tools like **pmchart**.
Instance domain refresh	The set of instances exported for each of the **trace** metrics can be cleared through the storable **trace.control.reset** metric.

15.7 Trace API

The **libpcp_trace** Application Programming Interface (API) is called from C, C++, Fortran, and Java. Each language has access to the complete set of functionality offered by **libpcp_trace**. In some cases, the calling conventions differ slightly between languages. This section presents an overview of each of the different tracing mechanisms offered by the API, as well as an explanation of their mappings to the actual performance metrics exported by the trace PMDA.

15.7.1 Transactions

Paired calls to the **pmtracebegin** and **pmtraceend** API functions result in transaction data being sent to the trace PMDA with a measure of the time interval between the two calls. This interval is the transaction service time. Using the **pmtraceabort** call causes data for that particular transaction to be discarded. The trace PMDA exports transaction data through the following **trace.transact** metrics listed in *Table 4.2. trace.transact Metrics*:

Table 4.2. trace.transact Metrics

Metric	Description
trace.transact.ave_time	The average service time per transaction type. This time is calculated over the last sample duration.
trace.transact.count	The running count for each transaction type seen since the trace PMDA started.
trace.transact.max_time	The maximum service time per transaction type within the last sample duration.
trace.transact.min_time	The minimum service time per transaction type within the last sample duration.
trace.transact.rate	The average rate at which each transaction type is completed. The rate is calculated over the last sample duration.
trace.transact.total_time	The cumulative time spent processing each transaction since the trace PMDA started running.

15.7.2 Point Tracing

Point tracing allows the application programmer to export metrics related to salient events. The **pmtracepoint** function is most useful when start and end points are not well defined. For example, this function is useful when the code branches in such a way that a transaction cannot be clearly identified, or when processing does not follow a transactional model, or when the desired instrumentation is akin to event rates rather than event service times. This data is exported through the trace.point metrics listed in *Table 4.3. trace.point Metrics*:

Table 4.3. trace.point Metrics

Metric	Description
trace.point.count	Running count of point observations for each tag seen since the trace PMDA started.
trace.point.rate	The average rate at which observation points occur for each tag within the last sample duration.

15.7.3 Observations and Counters

The **pmtraceobs** and **pmtracecount** functions have similar semantics to **pmtracepoint**, but also allow an arbitrary numeric value to be passed to the trace PMDA. The most recent value for each tag is then immediately available from the PMDA. Observation data is exported through the **trace.observe** metrics listed in *Table 4.4. trace.observe Metrics*:

Table 4.4. trace.observe Metrics

Metric	Description
trace.observe.count	Running count of observations seen since the trace PMDA started.
trace.observe.rate	The average rate at which observations for each tag occur. This rate is calculated over the last sample duration.
trace.observe.value	The numeric value associated with the observation last seen by the trace PMDA.
trace.counter	Counter data is exported through the **trace.counter** metrics. The only difference between **trace.counter** and **trace.observe** metrics is that the numeric value of **trace.counter** must be a monotonic increasing count.

15.7.4 Configuring the Trace Library

The trace library is configurable through the use of environment variables listed in *Table 4.5. Environment Variables* as well as through the state flags listed in *Table 4.6. State Flags*. Both provide diagnostic output and enable or disable the configurable functionality within the library.

Table 4.5. Environment Variables

Name	Description
PCP_TRACE_HOST	The name of the host where the trace PMDA is running.
PCP_TRACE_PORT	TCP/IP port number on which the trace PMDA is accepting client connections.
PCP_TRACE_TIMEOUT	The number of seconds to wait until assuming that the initial connection is not going to be made, and timeout will occur. The default is three seconds.
PCP_TRACE_REQTIMEOUT	The number of seconds to allow before timing out on awaiting acknowledgement from the trace PMDA after trace data has been sent to it. This variable has no effect in the asynchronous trace protocol (refer to *Table 4.6. State Flags*).
PCP_TRACE_RECONNECT	A list of values which represents the backoff approach that the **libpcp_trace** library routines take when attempting to reconnect to the trace PMDA after a connection has been lost. The list of values should be a positive number of seconds for the application to delay before making the next reconnection attempt. When the final value in the list is reached, that value is used for all subsequent reconnection attempts.

The *Table 4.6. State Flags* are used to customize the operation of the **libpcp_trace** routines. These are registered through the **pmtracestate** call, and they can be set either individually or together.

Table 4.6. State Flags

Flag	Description
PMTRACE_STATE_NONE	The default. No state flags have been set, the fault-tolerant, synchronous protocol is used for communicating with the trace PMDA, and no diagnostic messages are displayed by the **libpcp_trace** routines.
PMTRACE_STATE_API	High-level diagnostics. This flag simply displays entry into each of the API routines.
PMTRACE_STATE_COMMS	Diagnostic messages related to establishing and maintaining the communication channel between application and PMDA.
PMTRACE_STATE_PDU	The low-level details of the trace protocol data units (PDU) is displayed as each PDU is transmitted or received.
PMTRACE_STATE_PDUBUF	The full contents of the PDU buffers are dumped as PDUs are transmitted and received.
PMTRACE_STATE_NOAGENT	Interprocess communication control. If this flag is set, it causes interprocess communication between the instrumented application and the trace PMDA to be skipped. This flag is a debugging aid for applications using **libpcp_trace**.
PMTRACE_STATE_ASYNC	Asynchronous trace protocol. This flag enables the asynchronous trace protocol so that the application does not block awaiting acknowledgement PDUs from the trace PMDA. In order for the flag to be effective, it must be set before using the other **libpcp_trace** entry points.