Open Software Foundation
Request For Comments: 33.0
July 1995

R. Friedrich (HP)
S. Saunders (HP)
G. Zaidenweber (HP)
D. Bachmann (IBM)
S. Blumson (CITI)

STANDARDIZED PERFORMANCE INSTRUMENTATION AND INTERFACE SPECIFICATION FOR MONITORING DCE-BASED APPLICATIONS

1. INTRODUCTION

Distributed systems offer advantages in flexibility, capacity, price-performance, availability and resource sharing. Distributed applications can provide user productivity improvements through ease of use and access to distributed data. However, managing applications in a distributed environment is a complex task, and the lack of performance measurement facilities is an impediment to large-scale deployment.

This document describes performance instrumentation and measurement interface specifications that support performance related tasks such as configuration planning, application tuning, bottleneck analysis, and capacity planning. These performance measurement capabilities are a necessary component of any commercially viable computer technology, and are currently insufficient in DCE.

Specifically, to provide high-level analysis software the data to compute correlated resource utilization across nodes in a network, this document describes the:

(a) Functional specifications for a performance measurement access and control interface.

(b) Content of performance instrumentation within the DCE RPC runtime library and stubs.

(c) Extensions to support instrumentation of applications and other middleware technologies based on DCE.

The guiding philosophy is to define a set of *standardized performance instrumentation* that is consistently collected, reported and interpreted in a heterogeneous environment. Furthermore, these measurement capabilities are compiled into the core DCE services for use at customer sites. To support pervasive instrumentation the instrumentation must have minimal overhead on applications and services.

A companion RFC, RFC 32.0, discusses the requirements for performance monitoring, the metrics that are of interest for performance analysis and performance management, and the instrumentation necessary to collect performance data [RFC 32]. Consequently, the requirements for instrumentation are not described in this document.

1.1. Minimum Content for DCE 1.2

We recommend deployment of core instrumentation with the DCE Release 1.2 and then roll out additional instrumentation in later releases.

The following summarizes the minimum content for DCE release 1.2:

(a) Define and implement critical RPC **RTL** (runtime library, `libdce`) instrumentation.

(b) Define common access and collection interfaces for application servers and clients.

(c) Recompile/relink all DCE services to utilize the RPC instrumentation for Naming, Security, Time and DFS exported interfaces.

(d) Recompile/relink middleware with these instrumented DCE services.

(e) Link the performance measurement facilities (**observer** and **NPCS** (networked performance collection service, defined below)) with the standard instrumentation to allow monitoring the measurement system.

1.2. Terminology and Concepts

To ensure consistent meaning the following terms and concepts are defined for use in this document. A more detailed discussion of some of these concepts is found in later sections of this document.

1.2.1. Metrics

Metrics define measurable quantities that provide data to evaluate the performance of a system under study. They may consist of raw information (such as events) or derived quantities such as statistical measures or rates. Examples are response time, throughput, and utilization. These metrics, and more, are described in detail in section 4.

1.2.2. Instrumentation

Instrumentation are specialized software components incorporated into programs to provide mechanisms for measuring data that is used to calculate the relevant performance metrics. The basic measurement techniques are counting, timing and tracing. The objective of instrumentation is to provide measures of resource utilization (such as CPU, memory, I/O, network, etc.) and processing time (such as service time, queuing time, etc.). These measures are delivered to a **performance monitor** as statistical measures or as frequency and time histograms. From here on we will often refer to instrumentation as **sensors**.

1.2.3. Sensors

Sensors are the logical instantiations of the instrumentation necessary to collect data for a particular, single metric. Sensors consist of aggregations of **probes** located at well defined **probe points**. Sensors contain internal state that satisfies the definition of a particular metric. For example, a ‘‘response time sensor’’ will consist of two probes (a begin-timer and end-timer probe) but appear to the user as a single, logical entity. In object-oriented language, the sensors are the objects that encapsulate the data and functions provided by the instrumentation primitives.

A conceptual model of a sensor is illustrated in Figure 1. A sensor is a ‘‘software IC (integrated circuit)’’ that has input, output and control functions. The input to a sensor is provided by an event measured by a probe. The sensor provides output data, internal error conditions, and registration data so that the sensor can be identified by the measurement system. A sensor is controlled by several functions, including initialization, getting data, and modifying the sensor configuration. A sensor maintains internal state such as its identification, statistical data, and possibly some small algorithms that support threshold, histogram and trace functions.

There are three **types** of sensors:

(a) **Counter** sensors support the counting of events.

(b) **Timer** sensors support the timing of events (or functions).

(c) **Pass-thru** sensors support accessing data already available within a service that is not provided by a probe or allow arbitrary structures to be passed.

The first two sensor types support **threshold detection** to minimize data transmitted across the network by supplying data only when a user specified threshold criteria is met. All three sensor types support a **fast-path** option that does not set locks during sensor data update operations.

There are two **categories** of sensors, each of which support all three sensor types described above:

(a) **Standard** sensors are those defined and implemented by the core DCE services and are automatically available for an application with no source modifications. These sensors are statically-defined for a particular release of DCE.

(b) **Custom** sensors are specialized sensors created by application or middleware developers to count, time or pass-thru data specific to the application. Custom sensors are created within the process address space and are integrated within the measurement infrastructure.

Since DCE application environments are multi-threaded, all sensors must be re-entrant (in the case of custom sensors, this is the application-programmer’s responsibility). Sensors are described in detail in section 5.

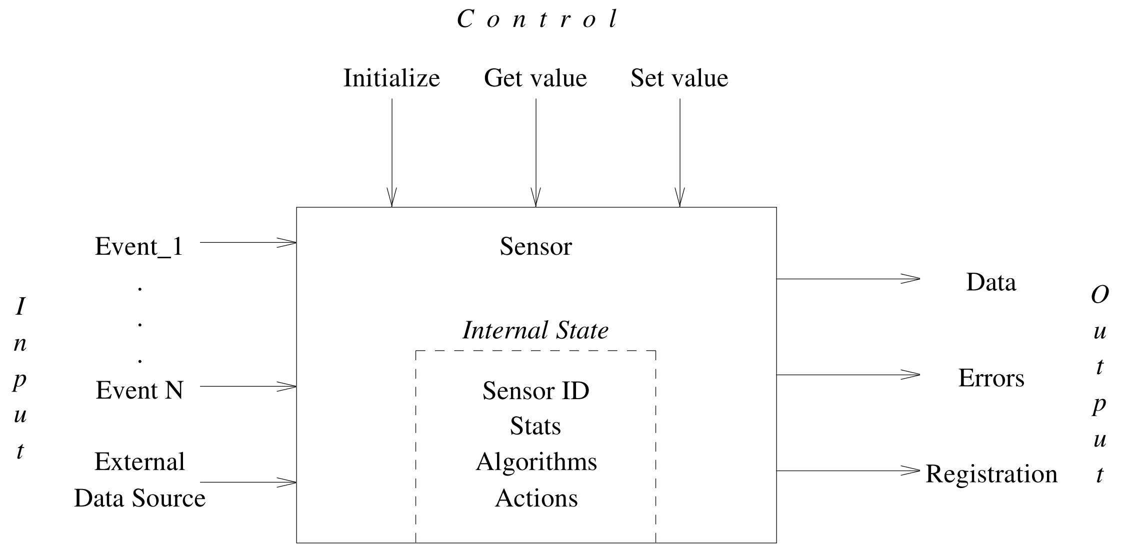

Figure 1. A sensor is conceptually illustrated here. A sensor can be thought of as a ''software IC'' that has input, control and output functions. In addition the sensor contains some internal state including sensor identifier, statistical metric data, metric computation algorithms, and other actions. The set of input, control and output functions are described in detail in sections 10 and 11.

1.2.4. Probes

Probes are the basic primitives from which sensors are constructed. Probes provide data input, control, and data access (output). For example, a probe might define the functions necessary to increment/decrement a counter. In general, probes do not contain local state, but only access global sensor data. (An exception is for timer probes, where the start-time must be maintained locally.) Probes are pre-defined as *macros* to ensure consistency in implementation of sensors and to ease instrumenting source code. The macro definitions are presented in section 6.

Probes provide input to a sensor. It is possible to place these probes in non-DCE services to obtain measures of interest (for example in the C library to collect data on sockets), but this spec focuses on DCE-based middleware and application software.

1.2.5. Probe points

Probe points are the locations within a program's flow of control where significant event transitions occur, and are thus candidates for the placement of probes. For example, when a client program issues an RPC a state transition occurs from ''user code'' to ''runtime library'', and is an excellent place for placing instrumentation software to record counts or elapsed times. The use of probes placed at probe points to construct a timer sensor is illustrated in Figure 2. Although the `probe_point_B` shown there is within the same scope as `functionN()`, it is not restricted to the same scope as `probe_point_A`.

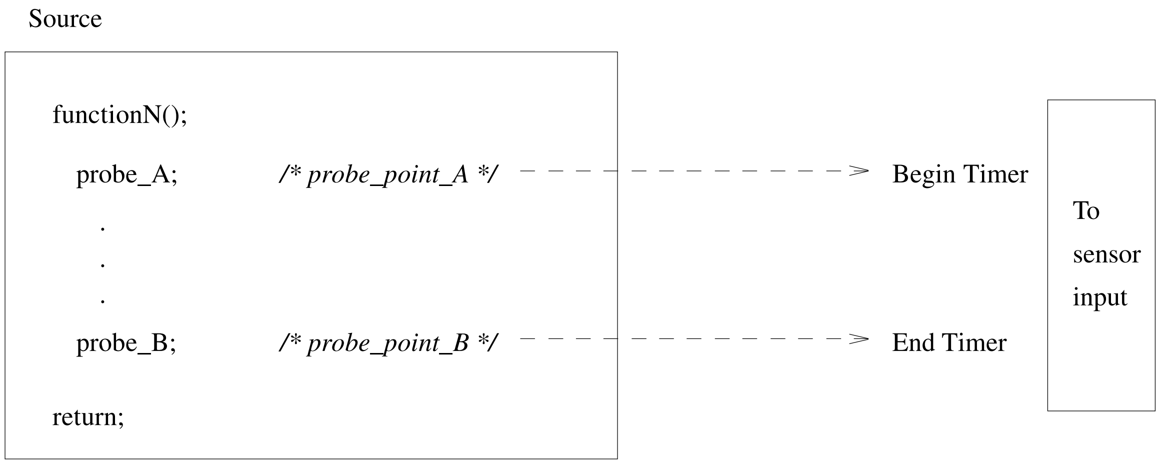

Figure 2. The implementation of a sensor timer is illustrated in this figure for an arbitrary `functionN()`. The probes are located at the beginning and ending of the function. These probe points provide input data into the sensor for starting and stopping an elapsed time clock.

1.2.6. Performance information sets

Requirements for different data capture granularities and subsets requires that the measurement system have a controllable capability to obtain only the required amount of data with minimum overhead. Consequently, we have defined varying data collection **information sets** to provide increasing detail in the collected data. This controls the detail (statistics) of the collected data. Under the best scenario there is no overhead incurred by the measurement system when no observations are required.

Increasing the size of performance information sets increases the number of data components of the collected data, providing a more comprehensive picture of operational behavior, but at the cost of increasing resource utilization. Information set control is done on a per-sensor, and not a per-process, basis.

Furthermore, for minimal overhead during continuous monitoring, metric **thresholds** are set, such that the measurement system will only report data when it exceeds the value of the specified thresholds. Minimizing resource consumption requires that **filtering** take place as close to the sensors as possible. This specification adopts the philosophy that the sensors themselves are simple and very efficient and that filtering tasks would complicate them needlessly. Consequently, filtering is done on the node, but by the NPCS.

Table 1 summarizes the sensor data information sets and their characteristics.

Info Set Value	Description	New Statistics Per Metric
0	Minimum overhead, no data needed.	None.
0x01	Provides simple utilizations, usage counts, error counts, mean times, mean rates ONLY if a user-specified threshold has been exceeded. Otherwise, no data is returned from the NPCS.	Counts, Simple sums, Minimums, Maximums.
0x02, 0x04, 0x08	Provides simple utilizations, usage counts, error counts, mean times, mean rates.	Counts, Simple sums, Minimums, Maximums.
0x10	Provides 2nd moments so that analysis can yield variance.	Sum of squares.
0x20	Provides 3rd moments so that analysis can yield skew.	Sum of cubes.

Table 1. Performance Information Sets

Event tracing is necessary to provide events in a time-ordered causal relationship. Due to scalability concerns and overhead in a production environment, this is not a part of the specification.

1.2.7. Reporting interval

The **reporting interval** is the time interval, measured in seconds, over which metrics are collected and statistics are summarized and then reported. To minimize performance measurement overhead, single events are not collected. Rather, the sensors summarize data over a reporting interval (currently 5 seconds minimum), and only report interval statistics to the higher level performance monitor. This interval is adjustable to decrease collection overhead.

1.2.8. Thresholds

Support of **threshold** sensors can dramatically reduce the amount of data collected and transmitted through the network environment, since only ‘‘exception cases’’ are reported. This supports the ‘‘management by exception’’ philosophy of network management. Thresholds are defined on a per-sensor basis, with a minimum and maximum (or both) values (i.e., a range). The NPCS then processes incoming sensor data, and when a sensor’s minimum is below the configured threshold the sensor data from this reporting interval is reported to the PMA (**performance monitor application**) at the next NPCS reporting interval. Supporting threshold detection in the NPCS simplifies the sensors and allows multiple PMAs to configure a specific sensor with different threshold values.

1.2.9. Network node

This document distinguishes the hardware from the software process for clients and servers. For the purpose of this paper, the physical hardware that clients and servers execute on is referred to as a **network node**. (Many management applications define a ‘‘server’’ as the hardware device that is providing the service. This is different from our definition).

1.2.10. DCE client

A **DCE client** is a software process/thread executing on a particular network node, that makes RPC requests. This definition includes a custom-developed application that issues RPC requests to a DCE server, as well as a DCE system-level service making a request of another DCE server.

1.2.11. DCE server

A **DCE server** is a software process/thread executing on a particular network node that receives (and usually responds to) RPC requests. This definition includes system-level DCE services (such as the `dced`) as well as custom-developed application services. Note that a ‘‘server’’ in this document is a software process and not the physical hardware (see definition of network node, above).

1.2.12. Performance monitor, and application (PMA)

A **performance monitor** (or just **monitor**) is a process that provides on-going collection and reporting of performance data for evaluation by system managers, application designers and capacity planners. A specific instance of a monitor that also supports management functions is called a **performance management application (PMA)**.

1.2.13. DCE Measurement System (DMS)

By the **DCE Measurement System** we mean the framework of of sensors, standard interfaces, and monitoring processes that initialize, control, access, and present performance data, as defined within this specification. Figure 4 in section 7 provides a block diagram for these components and their relationships.

The following processing elements are shown in figure 4:

(a) **Performance Management Application (PMA)** — The distributed application measurement system supports a single, logical view of the distributed application via a distributed application monitor. The most important views of the data provided by the PMA are discussed in more detail in section 1.3.2. It supports the **NPRI** interface. There are two special case PMAs: a **client-only PMA (COP)** that does not support the NPRI, and an **SNMP-agent PMA (SAP)** that interfaces with SNMP.

(b) **Sensors** are located throughout the application’s address space, and may reside in application and stub source code, and in libraries such as the DCE RTL. It is described in detail in section 5.

(c) **Observer** is a mechanism within the process’s address space that manages the sensors and optimizes the transfer of data outside the address space. It pushes the sensor data to the NPCS once per reporting interval, using the **PRI** interface. This library functionality resides within the DCE RTL and supports the PMI interface. It is described in detail in section 12.

(d) **NPCS** is the networked performance collection service. There is one per node, and it supports access and control requests for distributed application performance data over the heterogeneous network. It supports the NPMI and PRI interfaces. It is described in detail in section 12.

(e) **Encapsulated library** is the vendor-specific library that supports communication between the observer and NPCS. This library implements the platform-specific version of the standard PMI and PRI interfaces. It is described in detail in section 7.2.

The following standard interfaces are also shown in figure 4:

(a) **Networked Performance Measurement Interface (NPMI)** — The standard interface to a DCE-based node-level service (NPCS) for accessing and controlling performance data collected by the measurement system in a heterogeneous network. This interface is used to access and control sensor data from components of a distributed application and then construct correlated information about the application. A novel feature of the NPMI is that it supports locating *client* processes (that today are not locatable using standard DCE services). It is described in detail in section 8.

(b) **Networked Performance Reporting Interface (NPRI)** — The standard interface to a PMA, used by the NPCS for reporting sensor data. It is described in detail in section 9.

(c) **Performance Measurement Interface (PMI)** — The standard interface to a DCE-based service, for accessing and controlling performance data collected by the measurement system in a heterogeneous network. The interface is provided automatically by DCE for all DCE client and server processes. It is described in detail in section 10.

(d) **Performance Reporting Interface (PRI)** — The standard interface to a DCE-based node-level service (NPCS), for reporting sensor data collected by each process. It is described in detail in section 11.

1.3. A Vision of a Distributed Application Monitor

This section describes a vision of a performance measurement infrastructure that efficiently supports distributed application performance monitoring. It describes the need for a pervasive measurement infrastructure, the PMA presentation requirements, and the estimated design center impact.

1.3.1. Pervasive measurement infrastructure

The requirements for a distributed measurement system are described in detail in [RFC 32] and supplemented in section 3. The present section discusses a vision of a software system that realizes these requirements. The components of the measurement capability described later in this document satisfy the requirements of this vision of a monitor for distributed applications.

Performance instrumentation should provide data for various users and uses:

(a) System designers need data to understand complex, dynamic system behavior.

(b) Application designers must evaluate resource consumption of designs.

(c) System managers require data for system sizing and acquisition, monitoring performance goals service levels, load balancing and planning for future capacity.

(d) System analysts collect data to determine input parameters to application models.

(e) System vendors can use this data to evaluate workload demands on the services they provide.

From these users' perspectives, different vendor solutions should converge to provide a seamless, single, ''logical'' view of the behavior of the distributed environment. This demands that a distributed measurement system must collect heterogeneous data from all vendor systems (nodes) and present it for analysis in a consistent manner. Therefore the specification of a distributed measurement system must define a common set of performance metrics and instrumentation to ensure consistent collection and reporting across heterogeneous platforms, define standard APIs to ensure pervasive support in heterogeneous environments, and utilize self-describing data to ensure accessibility, extensibility and customizability of the measurement architecture in heterogeneous environments.

For ease of use the measurement system should support concurrent measurement system requests with different configurations and sampling intervals, allow enabling/disabling the instrumentation on a running system without disrupting an active application environment, and support custom application-defined metrics and instrumentation. Collected data should also be accessible by third-party performance monitors and application clients.

A performance measurement system, although not a system management service in and of itself, is an important aspect of any system management capability. Therefore, the measurement system should converge wherever possible with relevant measurement standards and node-based measurement facilities. It should also provide a closed feedback loop, so that changes in a distributed application environment are evaluated using the data collected by the measurement system.

The measurement system should provide a correlated view of resource consumption across heterogeneous network nodes. It should also provide an infrastructure for integrating disparate performance measurement interfaces from the host operating system, networking, and major subsystems in the distributed systems infrastructure.

Figure 3 illustrates our notion of a measurement infrastructure that is closely integrated with a distribution infrastructure. Instrumentation (depicted by measurement ''meters'') is dispersed throughout the software components. These components, when grouped in a logical manner, constitute a distributed application. The measurement system collects, transmits, reduces and correlates data from all relevant constituent components. These components include the distribution infrastructure (such as DCE), the host platform (an instrumented operating system such as HP-UX or AIX, or a non-instrumented operating system such as those found on PCs), other middleware components (such as Distributed Objects or Transarc's Encina transaction manager), as well as the application developed client and server code.

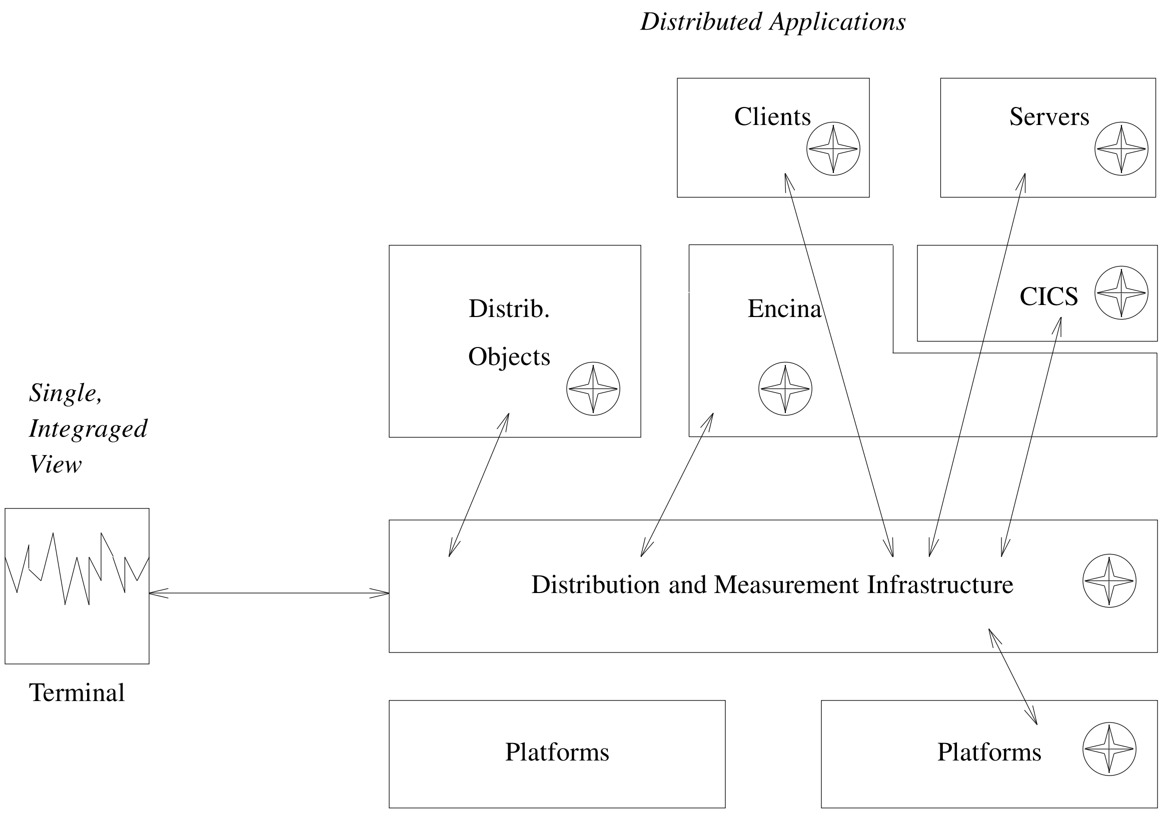

Figure 3. A measurement infrastructure for the performance monitoring of distributed applications. A well-designed measurement infrastructure should provide a ''centralized'' view of distributed objects and measure all aspects of the distributed application, not just the distribution infrastructure.

It is crucial to support a centralized view of the distributed application, regardless of the physical location of the components. For maximum flexibility, this centralized view is available from any node (assuming proper authorization). Finally, the instrumentation needs to provide a logical-to-physical mapping of the sensor names, as known by the user and stored by the measurement system.

The alternative to the approach illustrated in Figure 3 is to use several different performance tools, each running in a unique window, different for each platform in the network, presenting non-correlated and sometimes contradictory data. This approach is cumbersome, error-prone, inefficient, and ultimately useless, since distributed applications consist of interactions between logical groupings of software services. These logical groupings are impossible to capture and present without standardized instrumentation. Unfortunately, without standard performance instrumentation this is the only realizable alternative.

The efficiency of the infrastructure is important. If enabling performance monitoring excessively perturbs the environment then it is useless. The measurement system should minimize in-line overhead (the overhead in the direct dynamic path of the application) by deferring processing to outside of the application's direct path whenever possible. This technique still consumes CPU on the node, but minimizes the negative effect on application response time. Creating variable-size information sets (with increasing resource consumption) was described in section 1.2.6. Such variable information sets allow a person to ‘‘dial in’’ only the necessary monitoring data collection level (which minimizes overhead). A goal of the measurement system is to minimize network bandwidth consumed by the transmission of collected data. This is accomplished by summarizing data over intervals (instead of reporting every individual data item as it occurs), and supporting bulk retrieval interfaces.

Transmitted data may contain confidential information on application components or location, and requires a secure network communication channel to eliminate interception or modification.

In summary, standardized, pervasive performance instrumentation provides the following benefits:

(a) Supports monitoring of services on heterogeneous nodes.

(b) Ensures consistent metrics for interpretation.

(c) Provides fine grained view of server operations.

(d) Provides correlated views of client and server performance.

1.3.2. Possible PMA presentation views

The instrumentation and measurement system described by this RFC can provide data to support the following graphical and tabular presentation views of the PMA:

(a) **Summary Application View** — Display the response time and throughput of the application, by monitoring all or a subset of the application clients in the DCE Cell.

(b) **Summary Application Server View** — Display the response time, throughput, and CPU utilization of all or a subset of the application servers in the DCE Cell.

(c) **Summary Application View By Network Node** — Display the response time and throughput of the application, by monitoring all or a subset of the application clients executing on a particular network node.

(d) **Summary Application Server View By Network Node** — Display the response time, throughput, and CPU utilization of all or a subset of the application servers executing on a particular network node.

(e) **Component Application View** — Display the response time or throughput components of the application by monitoring all or a subset of the application clients in the DCE Cell.

(f) **Component Application Server View** — Display the response time, throughput, and CPU utilization components of all or a subset of the application servers in the DCE Cell. This includes fine-grain measurements at the level of per-interface summaries, and per-manager operation summaries.

However, a PMA is not required to support *all* of these views, or *only* these views.

2. SCOPE OF PROPOSAL

As we investigated the need and requirements for DCE performance instrumentation, we discovered that there exist several related activities and uses of performance data. How this specification incorporates these requirements is discussed in this section.

2.1. Scope

(a) **Performance Instrumentation** — The specific requirements for instrumentation are described in RFC 32.0 [RFC 32]. The requirements presented here supplement those outlined in RFC 32.0.

(b) **Managed Objects** — The DCE Management SIG group is defining a set of managed objects for the DCE [RFC 38]. We have reviewed their proposal and are working with the team to incorporate performance metrics into the managed object definitions.

(c) **Event Tracing** — The generalized tracing of events to collect performance data is an inheritantly non-scalable approach. Consequently it is not described in this document. A generalized event tracing mechanism for DCE is described in the RFC [RFC 11].

(d) **Computer Measurement Group PMWG Measurement Interface** — This group has proposed a standard OS performance measurement interface definition [CMG], and submitted it to X/Open. We support this effort but do not address it directly due to its current state as a submitted (as contrasted with accepted) X/Open draft.

(e) **Performance Management** — The instrumentation described herein forms the basis for a performance management system, but a management system *per se* is not described. That work should remain in the domain of management application products.

(f) **SNMP/CMIP and Network Management** — These techniques focus on network device management, in contrast to the application performance management described within this document. We support a ''polling'' function for the NPMI interface that can be used by an SNMP agent to collect performance measures from this instrumentation.

(g) **Accounting** — The instrumentation provides some data necessary for accounting purposes (such as charge-back), but does not describe an accounting system *per se*.

(h) **Fault/Error Detection** — Errors within an environment can have a serious performance impact, because of aborts or retries. The measurement system described here counts error conditions for RPCs.

2.2. Users

The following users of the performance instrumentation were identified.

2.2.1. Highest importance

(a) Operational and Administration Management.

Performance sensors should yield the critical information to enable dynamic control of a distributed application to improve its performance. Capacity planning and modeling are involved here as well since they utilize this data as input parameters.

2.2.2. Medium importance

(a) Resource Accounting (partially an auditing function; not only performance data needed here).

A goal is to provide resource consumption data that accounting requires, to eliminate redundant collection mechanisms. This proposal is not intended to be a competitive or complete mechanism for all of accounting's needs. Some information is outside of the capabilities described in this paper. (e.g., a strict accounting of ''which client called which server method'', and all the network, CPU, memory, and disk resources for that RPC).

(b) Tracing of Transactions and Events (for modeling or auditing).

Required for topology and application understanding. No event trace facility is provided by this proposal.

2.2.3. Lowest importance

(a) Detailed System S/W observation (tuning/troubleshooting).

There will always be a role for ''lab tools'', which by virtue of high overhead on the system or proprietary low-level nature, are not feasible in the production environment of an end-user. Lab tools will continue to exist but this specification does not explicitly address their requirements. However, this proposal does not preclude their use. Tools built on top of this proposed infrastructure can be used in the lab in providing basic information that is easily obtained (as does `vmstat()` and `iostat()` serve in some internal benchmarks for sanity checking).

3. MEASUREMENT SYSTEM REQUIREMENTS

The following are the basic requirements that we agreed are necessary for the success of this specification. When we ranked them, only a few were ranked less than ‘‘MUSTS’’.

(a) Extensibility of architecture:

  (i) Allow dynamic creation of new sensors.

  (ii) Extends to data store (self-describing data).

  (iii) Basic sensor types provide most functionality.

This specification does not aspire to recognize every sensor need that might ever be needed for distributed systems. As a result, the architecture must have extensibility as its core, to accommodate new sensors throughout its collection, naming, and display capabilities. As new applications are developed, middleware versions are released, or current runtime libraries are enhanced, the recognition of the need for additional sensors must be accommodated.

(b) Dynamic Control of sensors:

  (i) Enable/disable sensors (i.e., instrumentation can be dynamically disabled such that overhead is negligeable (˜ 0%), when sensors are off).

  (ii) Select amount of sensor data (sums, means, variance, histograms).

  (iii) Deliver sensors data periodically, or only at thresholds.

In the interests of operational efficiency, only the overhead associated with the currently required sensors should be imposed on the system. Even with a particular sensor, there needs to be the capability of providing simple sums or means when this information is sufficient, but also have the capability to supply higher statistical moments or distributions when necessary.

(c) Pervasive instrumentation:

  (i) No application source changes required for instrumentation.

  (ii) No application recompilation necessary to enable sensors.

  (iii) Environment is pre-populated with basic sensors.

This requirement assures the DCE customer that his/her application is monitorable, independent of the hardware platforms on which it is running.

(d) Measurements available in production systems:

  (i) Sensor overhead under strict architecture constraints.

  (ii) Dynamic control of sensors.

This requirement assures the DCE customer that his/her application is monitorable in a production system, since the architecture specification has strict guidelines to minimize overhead.

(e) Administration ease of handling sensor meta-data:

  (i) Naming, classification, and registration.

  (ii) Easily controlled sensor status.

  Sensors are more complex than simple counters. The architecture which prescribes their naming, organization and control is thereby critical to implementation and deployment.

(f) Consistency of sensor metrics:

(i) Definitions (agreement on specifics and names as described in RFC 32.0 [RFC 32]).

(ii) Results (all vendor implementations).

Pervasive instrumentation also requires consistently defined metrics, so that valid operations can be performed on sensors implemented in a heterogeneous environment.

(g) Security:

(i) Controlled access to interfaces.

(ii) Protected performance data on the network.

Provide user configurable access and data protection for sensor names and data.

(h) Validation Suite (at implementation):

(i) Adherence to the sensor performance spec.

(ii) ''Branding'' to conformance functional specification set.

Ensure that metrics are valid from release to release.

(i) Compatibility — Interplay with other performance tools:

(i) Higher importance:

[a] X/Open DCI.

(ii) Lesser importance:

[a] SNMP.

[b] 3rd party tools (e.g., PerfVIEW (HP) and Toolkit/6000 (IBM)).

Ease access to performance data for new and legacy application and system management tools.

4. PERFORMANCE METRICS AND STATISTICS

This section describes the metrics and statistics that guide the design and placement of performance instrumentation. Performance metrics are provided for a client perspective (end user) and for a server perspective. A detailed description of the sensors that collect these performance metrics is found in section 12.

4.1. Fundamental Performance Metrics

The following metrics define the quantities and the notation that are used throughout the remainder of the document. The metrics and notation have been derived from [Laz].

(a) **T** — The length of **time** that observations (measurements) were made.

(b) **A** — The number of request **arrivals** observed.

(c) **C** — The number of request **completions** observed.

(d) λ — The **arrival rate** of requests: $\lambda = \mathbf{A/T}$.

(e) **X** — The **throughput** of completions: $\mathbf{X} = \mathbf{C/T}$.

(f) **B** — The length of time that a single resource was **busy**.

(g) **U** — The **utilization** of a resource: $\mathbf{U} = \mathbf{B/T} = \mathbf{X}\times\mathbf{S}$.

(h) **S** — The average **service requirement** per request: $\mathbf{S} = \mathbf{B/C}$.

(i) **N** — The average **number of requests** in the system: $\mathbf{N} = \mathbf{X}\times\mathbf{R}$.

(j) **R** — The average system **response/residence time** per request.

(k) **Z** — The average user **think time**.

(l) $\mathbf{V_k}$ — The average number of **visits** that a system level request makes to resource **k**.

(m) $\mathbf{D_k}$ — The **service demand** at resource **k**: $\mathbf{D_k = V_k \times S_k = B_k/C}$.

(n) $\mathbf{Q_k}$ — The average **queue length** at resource **k**.

(o) $\mathbf{W_k}$ — The average **waiting time** at resource **k**.

(p) $\mathbf{L_k}$ — The average count of **locking contention** (unsatisfied lock requests) at resource **k**.

In general, a metric with an annotation of ‘‘**k**’’ is for a particular resource **k**. Non-annotated metrics are for the system as a whole. The above non-annotated metrics can also be defined for a particular resource. For example, ‘‘λ_k’’ is the arrival rate of requests at resource **k**.

4.2. Client Performance Metrics

The following metrics are collected or derived from a client perspective:

(a) Response time.

(b) Number of server request completions.

(c) Service demand.

(d) Think time.

(e) Number of active clients in system.

(f) Length of measurement interval.

4.3. Server Performance Metrics

The following metrics are collected or derived from a server perspective:

(a) Number of arrivals.

(b) Arrival rates.

(c) Number of completions (only non-error RPCs are counted).

(d) Throughput.

(e) Service requirement.

(f) Residence time.

(g) Visit count (includes error conditions).

(h) Waiting (queue) time.

(i) Queue length.

(j) Utilization.

(k) Measure of locking contention (count).

(l) Length of measurement interval.

4.4. Collected Statistics

The instrumentation must provide analysis software with the data required to compute the following statistical quantities:

(a) *Minimum*, during a sensor reporting interval.

(b) *Maximum*, during a sensor reporting interval.

(c) *Sum*, since sensor enabled for collection.

(d) *Mean*, since sensor enabled for collection.

(e) *Variance*, since sensor enabled for collection.

5. STANDARD AND CUSTOM SENSORS

This section describes how sensors are named in the cell, and their high level functions. The macro primitives used to construct these sensors are described in section 6. This section focuses on the standard (default) sensors in the distribution infrastructure (i.e., DCE), and custom sensors usable by other middleware technologies and application developers.

5.1. Sensor Naming

This section describes the semantics and syntax of sensor naming.

5.1.1. Terms of interest

Several terms are used in sensor naming and are described as follows:

(a) A **metric** is an abstraction without physical meaning, e.g., marshalling time. This is the concept of interest to the performance analyst.

(b) An **instance** is a physical manifestation of the metric, e.g., marshalling time for inbound parameters for interface `interface_0` and its `manager_operation_2()` operation.

(c) A **sensor** is the implementation that measures an instance of a metric in a particular process's address space on a particular host.

Consequently, metrics are not dynamic, but instances are. The dynamic instances are those aspects that may not be known at process link or load time, such as interface (since a server can register and unregister interfaces) or fileset (since filesets can be moved between DFS servers). The sensor name should have the dynamic elements as the suffix to allow naming into SNMP MIBs.

The full name of a sensor consists of three parts:

(a) The process name.

(b) The metric name.

(c) The instance.

The process name is used by the performance management application (the NPMI client) to locate the correct NPCS and tell it what sensors are of interest. The metric name and instance are converted by NPCS into the corresponding sensor identifier which is used to access the right sensor. The data structures that implement naming are described in section 7.3.2.

5.1.2. The process name

The process name identifies which process on which host is being queried. A process may have more than one name, e.g., a CDS server can be named by

```
/.:/hosts/dceperf.node101.osf.org/cds-server
```

as well as by

```
/.:/hosts/dceperf.node101.osf.org/perf-server/cdsd
```

or by

```
/.:/hosts/dceperf.node101.osf.org/perf-server/11345
```

A `dfsbind` (client-side DFS helper) could be named as

```
/.:/hosts/orion.node42.osf.org/perf-server/dfsbind
```

or

```
/.:/hosts/orion.node42.osf.org/perf-server/14316
```

The process name is used by the NPMI client to bind to the appropriate NPCS, thus any naming scheme that can be used by DCE clients to bind to DCE servers will work for NPMI clients as well. For current DCE implementations, that is the DCE Cell Directory Service (CDS). In the future this may be Federated Naming or other schemes.

The names used to specify a particular process to the NPCS can be either process IDs or executable names. The process ID is guaranteed to be unique, but requires first somehow finding out the ID, either by querying NPCS or other means. It may not have meaning on some platforms. The program name is more user-friendly, but may not be unique, especially in the case of clients on multi-user machines. The process ID is also more suitable for use by numeric naming schemes such as SNMP.

Both the process name and service name allow for continuity in time despite server restarts. They also avoid the problem of recycling of process IDs by the OS.

5.1.3. The metric name

The second part of the sensor name is the name of the particular metric (e.g., `rpc_calls`). The third part specifies the instance, e.g., protocol or interface and manager.

A metric has only one name, which is specified in this section for standard sensors, and made public via some similar mechanism for custom sensors. To avoid collisions these start with a domain identifier, where domain is the name of the DCE-based service domain (e.g., Encina, DFS, User, DCE, Security, ...). These domains should be registered with the OSF and documented in an OSF-RFC.

The metric name has two forms, a human readable list of slash-separated names (e.g., `dce/packets-out/protseq`), and a dot-separated list of numbers or **object ID (OID)** (e.g., `1.3.4`). These names are then suffixed with the name identifying the instance, giving, say, `dce/packet-out/protseq/ncadg_ip_udp` and `1.3.4.1`.

It is expected that users will typically specify a sensor by the human-readable name, while programs are more likely to use the object ID notation amongst themselves. Also, when SNMP agents are mapping the metric namespace into the MIB, the OID for the sensor will be the name used in the MIB.

For efficiency, the data provided by a sensor is treated as atomic, and any subparts are not nameable. The entire set of data is accessed as a whole via both the PMI and NPMI.

5.2. General Sensor Functions

This section describes functions supported by all sensors.

5.2.1. Fast-path

The fast-path option supports non-locking, to minimize update cost for those sensors where losing an update is considered acceptable. Note that this option cannot result in decreased reliability of a DCE process or service.

5.2.2. Information sets

Selectable statistical levels are supported for each sensor, namely, the minimum, maximum, sum, mean, and variance are collected, based on the collection information set.

5.2.3. Reporting interval

Selectable reporting interval allows modifying the interval (in seconds) that the sensor summarizes and reports data. Larger intervals reduce the amount of data transmitted across the network while reducing the granularity of the events measured. Summarization intervals will range from a minimum of 5 seconds to a maximum of 60 minutes.

5.2.4. Counter overflow

Counters are 32-bits (unsigned). This provides support for an activity that executes at the rate of 1.19 million operations per second for a maximum summarization interval of 1 hour. Overflow is a concern only if the counters value wraps twice in a single summarization interval. This is not likely. Consequently, overflow will be handled by the PMAs, since the data is cumulative and can be extracted. Sensors do not have to worry about overflow.

5.2.5. Threshold detection

Threshold detection and notification occurs for counter and timer sensors when a threshold condition is true. A threshold condition is a value range and a flag that specifies whether the threshold test should occur for values above or below this configured value. For example, a response time sensor set to detect thresholds would report data only when a user-configured threshold condition is true (for example, maximum response times are greater then 20 seconds). It is important to note that threshold detection is based on minimum or maximum values.

5.2.6. Minimum and maximum values

During a reporting interval, the minimum and maximum values are retained and returned. At the end of each reporting interval, the minimum and maximum are reset. This provides insights into the variation of the metric for a single interval (and not over the long term; it is a responsibility of the PMA to keep track of long term minimum and maximum behavior).

5.2.7. Histograms

Histograms provide distribution frequencies for a monitored event. They are not supported in this version of the specification, but are a candidate for future support.

5.2.8. Registration

Standard and custom sensors register with NPCS using the data structures and functions described in sections 7.3.2 and 6.2.

Custom sensors also require a utility to load their specific metric attributes into the DCE CDS for use throughout the cell. This utility is not defined by the specification.

5.2.9. Metric types

The specification defines a wide range of metric attributes that are described in detail in section 7.3.7.

5.3. Counter Sensors

Based on the client and server metrics described in sections 4.2 and 4.3, the following counter sensors are implemented for each client process and for each server RPC interface.

For each sensor the minimum, maximum, sum, mean, and variance are collected based on the collection information set.

5.3.1. Standard client counter sensors

(a) Calculate the *client RPC throughput rate*.

This measures the client's total RPC throughput rate as determined by the number of successful completions of client RPC requests per unit time.

Collect the data to compute the following:

(i) Total for all servers invoked.

(ii) Total by server.

(iii) Total by server-interface.

(iv) Total by server-interface-operation.

Note that throughput is a rate. The sensor keeps track only of request completions, thus higher-level software must divide this by the current measurement interval to compute the rate.

(b) Count the number of *RPC calls initiated* by the client.

This measures the frequency of client requests. Collected for each RPC server interface invoked by the client.

(c) Count of *total RPC packets sent* by the client.

This metric measures the number of packets sent by the client, and should be collected per protocol sequence (i.e., the number of packets passed to the network transport — not necessarily the number of network packets). Collected for each RPC server interface invoked by the client.

(d) Count of *total RPC packets received* by the client.

This metric measures the number of packets received by the client, and should be collected per protocol sequence (i.e., the number of packets passed to the network transport — not necessarily the number of network packets). Collected for each RPC server interface invoked by the client.

(e) Number of *total bytes sent per RPC call* from the client to the server.

This metric provides information about the size of the data transferred from the client to the server. Collected for each RPC server interface invoked by the client.

(f) Number of *total bytes received per RPC call* from the server to the client.

This metric provides information about the size of the data transferred from the server to the client. Collected for each RPC server interface invoked by the client.

(g) Count the *number of RPC call errors and failures*.

This information, although not a ''performance'' metric properly so-called, provides insight into the operational environment, and whether error conditions might be causing performance problems.

(h) Count the number of *lock request waits*.

Count the number of DCE thread lock requests that could not be satisfied, and so resulted in thread waits. Note that the lock path is a high-frequency, performance-critical path, and extra care must be employed to instrument it without resulting in a performance degradation.

(i) Count the number of server *binding lookup requests*.

Count the number of NSI (or, perhaps in the future, XFN) binding look-ups and imports. Collected for each RPC server interface invoked by the client.

(j) Count the number of *NSI entities returned*.

Count the number of NSI (or XFN) entities returned from look-ups and imports. Collected for each RPC server interface invoked by the client.

5.3.2. Standard server counter sensors

(a) Calculate the *server throughput* rate.

This measures the server's total RPC throughput rate, as determined by the number of successful completions of client RPC requests per unit time.

Collect the data to compute the following:

(i) Total by server.

(ii) Total by server-interface.

(iii) Total by server-interface-operation.

Note that throughput is a rate. The sensor keeps track only of request completions, thus higher-level software must divide this by the current measurement interval to compute the rate.

(b) Count of *total RPC packets sent* by the server.

This metric measures the number of packets sent by the server for all clients.

This metric should count packets sent by the server including nested RPCs sent to other servers. Collected for each RPC server interface.

(c) Count of *total RPC packets received* by the server.

This metric measures the number of packets received by the server for all clients.

This metric should count packets received by the server including nested RPCs received from other servers. Collected for each RPC server interface.

(d) Number of *total bytes sent per RPC call* from the server to the client.

This metric provides information about the size of the data transferred from the server to the client. Collected for each RPC server interface.

(e) Number of *total bytes received per RPC call* from the client to the server.

This metric provides information about the size of the data transferred from the client to the server. Collected for each RPC server interface.

(f) *Queue length* at the server.

This metric provides information about the queue length of RPC calls at the server, due to a lack of available call threads. This differs from calls queued (see next item), by providing a distribution of queue length.

(g) Count the number of *RPC calls queued* at the server.

This metric provides information about the number of RPC calls that were queued at the server, due to a lack of available call threads. This differs from queue length (see previous item) by providing only a count of calls queued.

(h) Count the number of *active call threads*.

This metric provides information about the utilization of the server's thread pool, by counting the number of active (non-idle) threads.

(i) Count the number of *RPC call errors and failures*.

This information, although not a ''performance'' metric properly so-called, provides insight into the operational environment and whether error conditions are causing performance problems.

(j) Count the number of *lock request waits*.

Count the number of DCE thread lock requests that could not be satisfied, and resulted in thread waits. Collected for each RPC server interface.

5.3.3. Custom counter sensors

The following custom sensors are available to the application developer to use for specific application events.

(a) *Counter sensor*.

This measures the total count of an application-specified event during the previous measurement interval.

5.4. Timer Sensors

Based on the client and server metrics described in sections 4.2 and 4.3, the following timer sensors are implemented for each client process and for each server RPC interface.

For each sensor the minimum, maximum, sum, mean, and variance are collected based on the collection information set.

5.4.1. Standard client timer sensors

(a) *Response time per RPC call* from the Client perspective.

This measures the total elapsed time, including server processing time and delay/queueing, for a client routine that invokes a particular DCE server.

Collect the following data:

(i) Total for all servers invoked.

(ii) Total by server.

(iii) Total by server-interface.

(iv) Total by server-interface-operation.

Measure the elapsed time per RPC call, from the time the client's runtime initiates the call until the last packet has been received by and unmarshalled at the client. This should include nested RPC call elapsed times if other DCE servers, such as the security service, are invoked (the nested RPC call time is optionally broken out). RPCs that result in DCE errors should be reported in a separate category, not included in this one.

Note that this time will not include client application or user interface response time, since those are outside (‘‘above’’) the scope of the DCE services.

(b) *Service requirement at client* for all RPCs.

This measures the service requirement at the client, including operating system and network software CPU processing time, required to satisfy a client's RPC request. This request may consist of multiple RPC packets, but only one RPC call. This requires that the host operating system support a performance measurement system and that DCE servers use it to gather CPU service time. The implementation of this sensor is thus host OS dependent. Data is collected on a per-server interface.

(c) *Marshalling time at client* for all RPCs.

This measures the marshalling time of RPC parameters at the client required to satisfy a client's RPC request. Data is collected on a per server interface.

(d) *Unmarshalling time at client* for all RPCs.

This measures the unmarshalling time of RPC parameters at the client required to satisfy a client's RPC request. Data is collected on a per server interface.

(e) *RPC network delay*.

This measures the delay of the network between a particular client and server node, as measured between client and server runtime libraries. Consequently, it measures the latency of the networking software transport, in addition to the physical network wire. The data is collected per transport protocol sequence. (DTS may already capture this ‘‘DCE ping’’ time, and if so, then it should be used.)

5.4.2. Standard server timer sensors

(a) *Residence time per RPC call* from the Server perspective

This measures the total elapsed time, including server processing time and delay/queueing, required for the server to satisfy a client request.

Collect the following data:

(i) Total by server.

(ii) Total by server-interface.

(iii) Total by server-interface-operation.

Measure the elapsed time per RPC call, from the time the server runtime receives the call until the last packet has been marshalled by the server and sent. This should include nested RPC call elapsed times if other DCE servers, such as the security service, are invoked (the nested RPC call times are optionally broken out). RPCs that result in DCE errors should be reported in a separate category, not included in this one.

Note that the elapsed time does not begin to accumulate until a thread from the call-thread pool is dispatched on behalf of this incoming request; consequently, this does not include call-thread queueing time prior to the first call thread dispatch. This queuing time is collected by the initial queuing time at the server.

(b) *Initial queueing time at server* for all RPCs.

This measures the queueing time of an incoming RPC request if no call-thread is available to dispatch. See residence time (previous item) for complementary elapsed measure time.

(c) *Service requirement at server* per client request.

This measures the service requirement at the server, including operating system and network software CPU processing time, required to satisfy a client's request. This request may consist of multiple RPC packets, but only one RPC call. This requires that the host operating system support a performance measurement system and that DCE servers use it to gather CPU service time. Data collected on a per-server-interface-operation basis.

(d) *Marshalling time at server* for all RPCs.

This measures the marshalling time of RPC parameters at the server required to satisfy a client's RPC request. Data collected on a per-server interface operation basis.

(e) *Unmarshalling time at server* for all RPCs.

This measures the unmarshalling time of RPC parameters at the server required to satisfy a client's RPC request. Data collected on a per-server-interface-operation basis.

(f) *Interarrival time at server* for all RPCs.

This measures the interarrival time of incoming RPC requests. Data collected on a per-server-interface-operation basis.

5.4.3. Custom timer sensors

The following custom sensors are available to the application developer to use for specific application events.

(a) *Timer sensor*.

This measures the total elapsed time, including processing time and delay/queueing, for an event as determined by the application developer.

5.5. Pass-thru Sensors

Custom sensors can be defined that pass opaque data through the measurement systems. These sensors merely copy data from existing internal data structures. These sensor data types are opaque, and require pickling routines for support which are supplied at sensor registration time.

The DCE 1.1 IDL compiler supports ‘‘pickling’’, i.e., support for encoding and decoding data types to and from a byte stream format. A sensor may take advantage of this pickling process to encode data into the opaque array of bytes, which it is able to transmit via the standard interfaces. This allows sensors to be created with elaborate data types and provides a mechanism for that data to be marshalled.

5.6. Standard Operating System Sensors

Collecting host-specific resource consumption (such as service demand) requires accessing the host operating system's measurement system. Specifically, each DCE host's operating system should provide the following application specific metrics via a standard interface:

(a) CPU utilization (system + user).

(b) File Disk I/Os per second.

(c) Paging I/Os per second.

(d) Network packets per second.

(e) OS dispatcher queue length and average queue time.

(f) Process physical main memory usage.

(g) Process virtual memory usage.

These host OS performance metrics can be reported by the observer as a process global metric. The X/Open DCI [CMG] is a good candidate to provide a standard interface to operating system measures. If the host OS does not support the DCI, then these sensors will require porting to the proprietary OS measurement interface.

6. SENSOR PROBE MACROS AND FUNCTIONS

This section describes the macros that are used at various probe points to construct sensors. These software probes, implemented as a set of macros, implement each sensor to ensure consistency and decrease implementation time for DCE developers and application writers.

6.1. Sensor Data Flow

During process initialization, various process-wide sensors, such as `rpc_call_thread_utlization` and `rpc_queue_utilization`, are initialized and registered with the observer, using the functions in section 6.2.

The sensors associated with specific server interface operations are not registered until the server registers this interface via the RTL call to `rpc_server_register_if()`. Probes defining these sensors are located in the execution path of the RPC and store their data into a structure that ‘‘travels’’ with the RPC call. At the end of the call, after the call response has been sent to the client, all probe data is ‘‘tallied’’, and the global sensor data structure is updated. Some sensors are updated directly by the probe that executes during the event being sensed.

When the observer thread executes it checks for entries on its tally queue and updates those sensors. Then it searches the lists of registered sensors and builds a batch of updates to send to the PRI.

6.2. Sensor Registration and Data Functions

Functions for registering, unregistering sensors and queueing sensor data are described in this section.

```
/* These function-pointer definitions allow a subsystem
 * designer to provide callbacks to the observer for
 * controlling a subsystem and its sensors.  The functions
 * which are referenced must be re-entrant, as the code
 * updating the sensors and/or subsystems from the
 * middleware/application will be asynchronous from the
 * observer.  Each function defines a pointer to a control
 * block defined by the function writer as an [in] parameter,
 * and a 32-bit DCE format status value as an [out] parameter.
 * These may be passed in as NULL values, but this will prevent
 * any control information from being passed back up to the
 * subsystem/sensor from PMAs.
 */

typedef void (*dms_subsys_ctl_fn_t)
                              (void *ctlblock, unsigned32 *st);

typedef void (*dms_sensor_ctl_fn_t)
                              (void *ctlblock, unsigned32 *st);

typedef void (*dms_data_pickle_fn_t)
                                  (void *data, unsigned32 *st);
```

```
/* This structure contains information about a subsystem which
 * the observer may use to construct its persistent storage --
 * it's patterned from the information needed for an RPC
 * interface, but may be used for any type of subsystem defined
 * by a middleware or application designer.  Note the
 * presumption that all operations have the same properties and
 * are instrumented with the same number of sensors per
 * operation. This functionality is for batching registrations.
 * Sensor registration may be performed individually.
 *
 * The array of sensor descriptors is defined with dimension 1
 * to accommodate certain compiler limitations.  Nonetheless,
 * the array may be allocated at any size.  For example, one
 * may allocate an appropriately sized subsystem descriptor
 * with the following malloc call:
 *
 *   ssd = (dms_subsys_descriptor_p_t) malloc (
 *    (size_t) (sizeof(struct dms_subsys_descriptor) +
 *    (n_ops * sizeof struct dms_sensor_descriptor)
 *    ));
 *
 * The array does not need to be null-terminated.
 */

typedef struct dms_subsys_descriptor {
        uuid_t          subsys_uuid;
        void            *subsys_handle;
        dms_subsys_ctl_fn_t     ctl_fn;
        int             n_ops;
        int             n_sensors_per_op;
        char            *subsysname;
        dms_sensor_descriptor_t sensors[1]
        } dms_subsys_descriptor_t, *dms_subsys_descriptor_p_t;

/* This structure contains information about individual sensors
 * which the observer needs to construct its persistent storage
 * of sensor data and for registering sensors through the PRI.
 * These structures may be chained into the sensors field of
 * the subsystem descriptor to batch sensor registrations.
 *
 * The following fields may be set to 0 (or NULL) to disable
 * the respective functionality:
 *   ctl_fn
 *   millisec
 *   attrs
 */
```

```
typedef struct dms_sensor_descriptor {
        uuid_t          sensor_id;
        void            *sensor_handle;
        int             op_num;
        dms_sensor_ctl_fn_t     ctl_fn;
        char            *sensorname;
        int             millisec;
                        /* sampling interval;
                           0 if event-sampled */
        dms_data_descriptor_p_t sensor_data;
        void            *attrs[dms_HIGHEST_ATTRIBUTE]
        } dms_sensor_descriptor_t, *dms_sensor_descriptor_p_t;

/* The following structure is for describing a sensor’s data
 * format.
 */

typedef struct dms_data_descriptor {
        size_t          datasize;
        void            *data;
        dms_data_pickle_fn_t    data_fn
        } dms_data_descriptor_t, *dms_data_descriptor_p_t;

/* For registering interfaces or custom subsystems. */

void dms_obs_register_subsys (
        dms_subsys_descriptor_t *subsys,
        void            **subsys_handle,
        unsigned32      *st
        );

/* Opposite of register_subsys. */

void dms_obs_unregister_subsys(
        void            *subsys_handle,
        unsigned32      *st
        );

/* For registering sensors. */

void dms_obs_register_sensor(
        dms_sensor_descriptor_t *sensor,
        void            *subsys_handle,
        void            **sensor_handle,
        unsigned32      *st
        );

/* Opposite of register_sensor. */

void dms_obs_unregister_sensor(
        void            *sensor_handle,
        unsigned32      *st
        );
```

```
void dms_obs_queue_data(
        void            *sensor_handle,
        dms_sensor_descriptor_t *sensor,
        unsigned32      *st
        );
```

6.3. Sensor Probe Macros

This section describes the probe macros used to create sensors. For each macro, only the function signature (pseudo-prototype) is provided. The macro body has been excluded in the interest of brevity. Note that the sensor data location is passed into each relevant macro.

```
/* Utility functions: Zero-out the values in a timestamp
 * Pseudo-prototype:
 *  void DMSTIMEZERO(struct dms_timestamp *);
 */

/**************************************************************
 * For those cases where interval times are deemed more
 * appropriate, the following data and macro definitions may be
 * used.
 */

/* An interval timer data structure allows preservation of both
 * begin and end timestamps, returning the interval in a new
 * timeval structure.
 */

typedef struct dms_itimer {
        struct dms_timestamp    intervalstart;
        struct dms_timestamp    intervalstop;
        struct timeval          interval;
        } dms_itimer_t;

/* Start interval timer
 * Pseudo-prototype:
 *  void DMS_INTERVALSTART(struct dms_itimer);
 */

/* Stop interval timer, and calculate wallclock time
 * Pseudo-prototype:
 *  void DMSINTERVALEND(struct dms_itimer);
 */

/**************************************************************
 * Counter and MIN/MAX Probe Data structures
 */

/* Counter element. */

struct dms_probe_cnt {
        long    counter;  /* local value maintained by probe */
        };

/* Minimum/Maximum element. */
```

```
struct dms_probe_mm {
        int     reset;          /* reset command from sensor */
        unsigned long   value;  /* value maintained by probe */
        unsigned long   *datum;   /* ptr to comparison datum */
        };

/* A pass-through probe datatypes: to be used for sensing
 * counters and/or timers (in gettimeofday() format) and/or
 * amorphous data chunks maintained elsewhere.
 */

struct dms_probe_vpt {
        unsigned long   localval;
                        /* local value maintained by probe */
        unsigned long   *value;
                        /* pointer to value fetched by probe */
        };

struct dms_probe_tpt {
        struct timeval  localval;
                        /* local value maintained by probe */
        struct timeval  *value;
                        /* pointer to value fetched by probe */
        };

/***************************************************************
 * Counter Probe.
 *
 * This probe will add any value to its counter. The second
 * argument may be a reference to a delta value maintained
 * elsewhere or to a constant.
 */

/* Pseudo-prototype:
 * void CNTPINIT(struct probeCounter A);
 */

#define CNTPINIT(A)        (A).counter = 0;

/* Pseudo-prototype:
 *  void CNTPROBE(struct probeCounter A, long valp);
 *
 * This probe may need to be protected by an appropriate mutex,
 * but is often used in conjunction with another probe also
 * needing the same mutex lock. Therefore, the code
 * instantiating this macro is responsible for explicitly
 * locking and unlocking the appropriate mutex if desired.
 *   RPC_MUTEX_[UN]LOCK((X)->m);
 */
```

```
/* Minimum/Maximum probes.
 *
 * These probes store the minimum [maximum] value of their
 * current value and a value stored elsewhere at the time they
 * execute.
 *
 * They are implemented to allow resetting.  The process for
 * resetting utilizes a "reset flag" in the probe structure.
 * When the controlling thread, usually the observer or a
 * thread under it's control, wants to reset the probe, it
 * unconditionally writes a non-zero value to the reset flag.
 * When the probe actually executes it checks this flag for
 * non-zero and branches based on its value:
 *   If zero, it executes the minimum [maximum] function.
 *   If non-zero, it sets the data value to the current value
 *    of the data and then clears the reset flag.  Once the
 *    reset flag is clear, the controlling thread may consider
 *    the data valid again.
 * This procedure is designed to minimize exposure to a case of
 * multiple threads trying to write data to the value location,
 * resulting in lost data.
 */

/* Pseudo-prototype:
 *  void MAXPINIT(struct probeMinMax A, long *datp);
 *  'datp' points to a long which is the comparison value in
 *  this and the following probes.
 */

/* Pseudo-prototype:
 *  void MAXPINIT(struct probeMinMax A, long *datp);
 */

/* Pseudo-prototype:
 *  void MAXPRESET(struct probeMinMax A);
 */

/* Pseudo-prototype:
 *  void MINPRESET(struct probeMinMax A);
 */

/* The minimum probe will store the minimum of its present
 * value and the datum it is sensing to its value.  The maximum
 * probe simply reverses the comparison clause of the ternary
 * operation.  The value is an unsigned long, the datum is a
 * pointer to unsigned long.
 */
```

```
/* Pseudo-prototype:
 *  void MAXPROBE(struct probeMinMax A);
 *
 * This probe may need to be protected by an appropriate mutex,
 * but is often used in conjunction with another probe also
 * needing the same mutex lock.  Therefore, the code
 * instantiating this macro is responsible for explicitly
 * locking and unlocking the appropriate mutex if desired.
 *   RPC_MUTEX_[UN]LOCK((X)->m);
 */

/* Pseudo-prototype:
 *  void MINPROBE(struct probeMinMax A);
 *
 * This probe may need to be protected by an appropriate mutex,
 * but is often used in conjunction with another probe also
 * needing the same mutex lock.  Therefore, the code
 * instantiating this macro is responsible for explicitly
 * locking and unlocking the appropriate mutex if desired.
 *   RPC_MUTEX_[UN]LOCK((X)->m);
 */

/* Pseudo-prototype:
 *   void PASSPROBE(dms_probe_vpt)
 * The function of this probe macro is to snapshot a dynamic
 * value stored outside the context of the DMS to a local value
 * in order to lessen concurrency issues and hopefully provide
 * more stable readings. Its use is not mandatory.
 *
 * This macro should work fine for either value or time
 * pass-throughs.
 *
 * This probe may need to be protected by an appropriate mutex,
 * but is often used in conjunction with another probe also
 * needing the same mutex lock.  Therefore, the code
 * instantiating this macro is responsible for explicitly
 * locking and unlocking the appropriate mutex if desired.
 *   RPC_MUTEX_[UN]LOCK((X)->m);
 */
```

6.4. Sensor Timer Functions

Timestamps play a crucial role in instrumentation but can also have high overhead. To resolve this the specification has defined several high-speed timer functions.

```
/**************************************************************
 * TIME functions.
 *
 * The DCE runtime maintains a correlation between of the value
 * returned by dms_gettime() with that returned from
 * gettimeofday().  The clocks should be presumed to be stable
 * and accurate and to remain exactly correlated over the
 * periodic re-correlation interval.  The re-correlation
 * interval should be a fairly small fraction of the
 * dms_gettime() wrap interval.  For instance, a 200 MHz
 * machine for which the time is maintained as a 32-bit value
 * of system clock ticks will wrap in about 20 seconds.
 *
 * We recommend a re-correlation interval of 5 seconds.  This
 * should be a small enough fraction of the wrap time, yet
 * infrequent enough to avoid unnecessarily increasing the
 * gettimeofday() overhead.
 */

#include <limits.h>

/* The following should be available from <limits.h>. */

#ifndef ULONG_MAX
# define ULONG_MAX      0xFFFFFFFFUL
#endif

#ifndef UINT_MAX
# define UINT_MAX       0xFFFFFFFFU
#endif

#ifndef INT_MAX
# define INT_MAX        0x7FFFFFFF
#endif

#define USEC_PER_SEC    1000000

typedef unsigned long dms_time_offset_t;

typedef struct dms_timestamp {
        struct timeval          base_wallclock;
        dms_time_offset_t       base_ticks;
        dms_time_offset_t       current_ticks;
        } dms_timestamp_t;
```

```
/************************************************************
 * DMS_TIMESTAMP() retrieves the information necessary for
 * computing an accurate timestamp (later) without calling
 * gettimeofday() inline.  It is structured to preserve the
 * information which will be required for later, out-of-line
 * calculation of time intervals. This macro must be passed a
 * valid pointer to struct dms_timestamp.
 * Pseudo-prototype:
 *  void DMS_TIMESTAMP(struct dms_timestamp *);
 */

/************************************************************
 * DMS_TICKS_TO_USEC() converts system-clock ticks to
 * microseconds.  This macro must be passed a valid
 * dms_time_offset_t. It is not normally invoked directly by
 * user code.
 * Pseudo-prototype:
 *  unsigned long DMS_TICKS_TO_USEC(dms_time_offset_t);
 */

/************************************************************
 * DMS_TS_TO_TV() converts the time stored in a dms_timestamp
 * structure to the format of timeval. Both input pointer
 * parameters must be valid.  It is not normally invoked by
 * user code.
 * Pseudo-prototype:
 *  void DMS_TS_TO_TV(struct dms_timestamp *, struct timeval *)
 */

/************************************************************
 * DMS_SUB_TIME() returns the difference between two timestamps
 * into a timeval structure
 * Pseudo-prototype:
 *  void DMS_SUB_TIME(
 *     struct dms_timestamp *,
 *     struct dms_timestamp *,
 *     struct timeval *);
 * If the timestamp for the end time is earlier than the
 * timestamp for the begin time, this macro will compute a
 * negative interval which may cause problems. Therefore, the
 * caller must check for the error condition (negative seconds
 * field -- the microseconds field is unsigned).
 */

/* DMS_GETTIMEOFDAY() fills in a struct timeval with the "real,
 * current" wallclock time without calling gettimeofday().
 * Pseudo-prototype:
 *  void DMS_GETTIMEOFDAY(struct timeval *);
 * This macro requires a valid pointer-to-struct-timeval.
 */
```

```
/*************************************************************
 * dms_gettime_int() is a fast, implementation-specific
 * function which returns an unsigned long with a
 * machine-dependent resolution.  Each implementor must provide
 * this system-specific function and the conversion factor
 * specifying the relationship of this number to a standard
 * time unit such as seconds or microseconds.
 */

extern dms_time_offset_t dms_gettime_int(void);
```

7. STANDARD INTERFACES

To achieve pervasiveness in a heterogeneous environment, the measurement system must support standardized interfaces that support access and control of both server and client sensors. This section provides an overview of the standard application programming interfaces (API), data structures and related capabilities. The ‘‘official’’ IDL files are located in appendices, and they supercede the discussion in this section.

7.1. API Overview

The standard interfaces of this spec provide the framework for inter-node and intra-node DCE performance instrumentation control and data transfer. Four APIs provide for the relationships diagramed in Figure 4 for each node in a DCE cell.

These four interfaces are the:

(a) **PMI** — Performance Measurement Interface, to the sensors contained within a process, used by the NPCS.

(b) **PRI** — Performance Reporting Interface, used by the observer to report sensor data to the NPCS.

(c) **NPMI** — Networked Performance Measurement Interface, to the NPCS, used by the PMA to communicate with the NPCS.

(d) **NPRI** — Networked Performance Reporting Interface, used by the NPCS to send sensor data to the PMA in bulk.

There are two categories of APIs. First, there are APIs at the DCE process level (the PMI and PRI); second, APIs at the node (machine) level (the NPMI and NPRI). The NPMI and the NPRI are used by the PMA developer. The PMI and PRI are used by the DCE vendor and the NPCS developer.

The NPMI provides the interface between the NPCS and any Performance Management Applications (PMA’s) that wish to access DCE performance instrumentation. The PMI provides the interface between the NPCS and DCE client and server processes. These processes contain the performance instrumentation, sensors. Basically PMA’s use the NPMI to discover and request/receive data from sensors. The NPCS uses the PMI to gain knowledge of DCE client and server processes, control the configuration of sensors, and receive data from sensors.

The NPMI and NPRI interfaces are RPC interfaces to leverage security and naming features of DCE. The PMI and PRI are node-local and can use any relevant IPC mechanism, including RPC, implemented in the encapsulated library described in section 7.2.

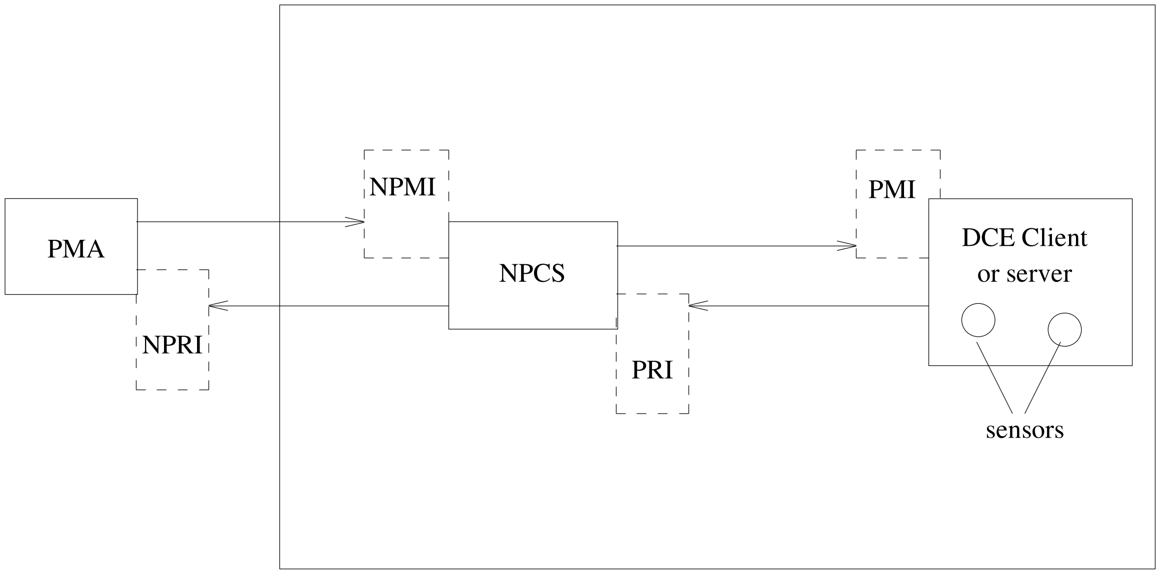

Figure 4. NPRI, NPMI, NPCS, PMI, PRI and sensor relationships.

Also, the NPCS is shown as an independent mechanism. Whether it is an independent process or part of another process is implementation-specific.

The NPMI, PMI, NPCS, and sensors exist and operate to provide PMA's with DCE performance instrumentation in the manner described below. The PRI and NPRI provide the communication channel to efficiently return sensor data to the PMA using a push protocol.

During the steady-state, runtime sensors collect specific metrics within the DCE environment whenever a thread executes their set of probes. Probes are the (inline) code sequences that capture the data needed to produce a metric, e.g., timestamps for a response time metric. This relationship is illustrated in Figure 5. During the execution of a distributed application, the flow of control passes from the client code into the client stub into the DCE runtime library (RTL1), possibly across a network, into the DCE runtime library (RTL2), into the server stub and into the server code. The thread of execution returns in a reverse manner. As it passes through RTL2 it encounters two probes, a begin-response-probe and an end-response-probe. After it passes through the end-response-probe the appropriate sensor is located and updated.

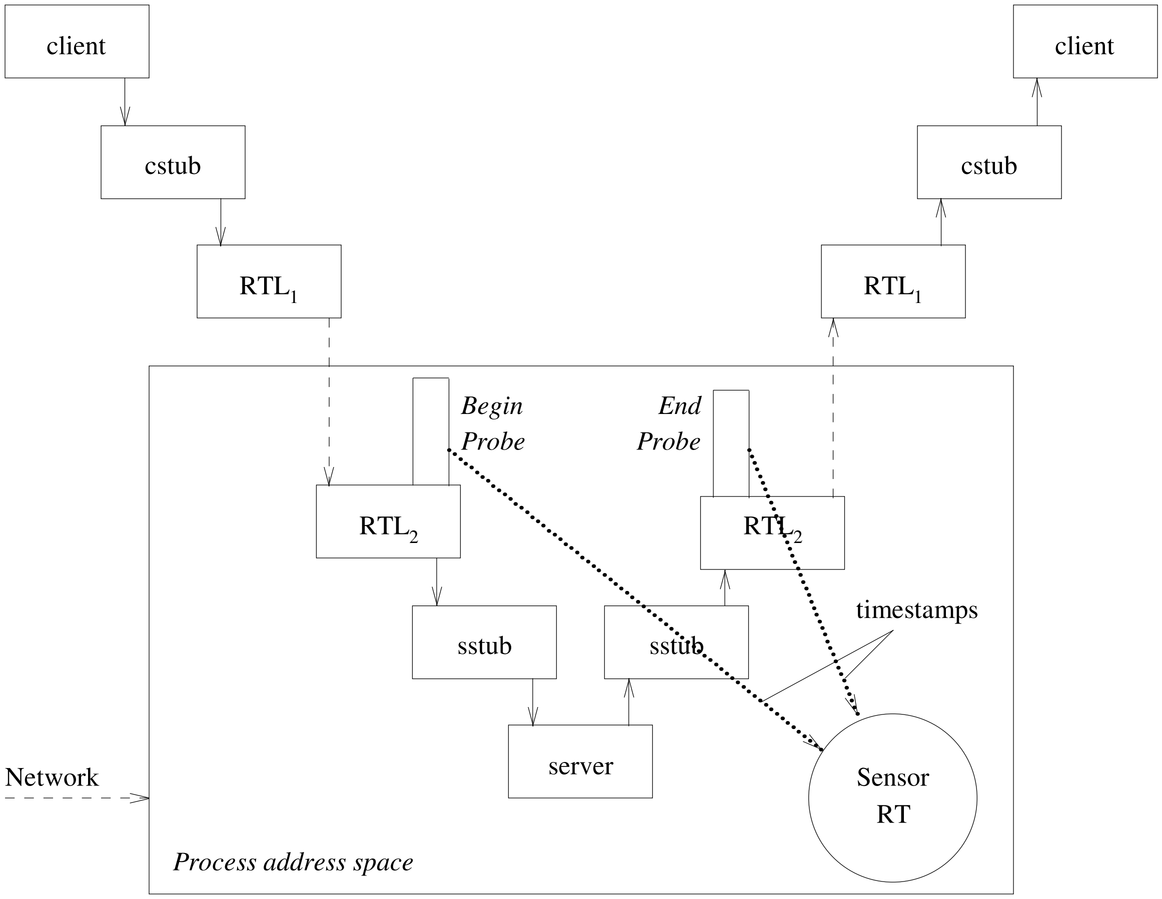

Figure 5. Flow of control, probes and sensors shown for a response time sensor in the DCE run time library. Probes are not restricted to the RTL and can also occur in client or server source or stubs.

Sensors provide metrics by supplying the component values necessary to calculate intervalized metric values in probes and store sensor data in a process accessible structure. The component values provided by sensors are in the form of cumulative totals, for example.

A sensor with the purpose of providing a response time metric (ignoring location) would make available a total number of responses (R), and a total of the time spans to produce those responses (RT). These values could be taken from the sensor at the beginning and ending of a time interval and the mean response time for that interval.

The observer (also known as the ''address space helper thread'') periodically captures the metric component values for each sensor that has been configured. The capture periodicity is specific/unique to each sensor. The observer will then communicate the captured metric component values and a timestamp to NPCS through the PRI interface.

The NPCS provides a consistent node-level view of all DCE performance instrumentation on a given node. It maintains a registry of sensors and observers provided to it through the PRI interface. It responds to queries against that registry made through the NPMI interface. It maintains a (single) copy of the latest captured metric component values for all registered sensors, communicated to it through the PRI interface. It maintains a registry of the collection of sensors that each PMA has configured through the NPMI interface. Based on the configurations requested by all PMAs, NCPS configures individual sensors through the PMI interface. It communicates the component metric values of any sensors that have been active during the requested (PMA-specific) interval through the NPRI interface.

7.2. Encapsulated Library

The connection between the NPCS and the instrumented DCE processes is a critical one; it is very high volume, so its performance is a major factor in minimizing the impact of instrumentation on the overall performance of a node. Because of this, the connection is specified as two interfaces whose implementation is deliberately left vendor-specific; the goal is to allow full use of any available system-specific mechanisms to minimize the overall cost of transfers. The central focus is on the actual reporting of collected data, since this will be the greatest volume and the most likely to occur during normal operation.

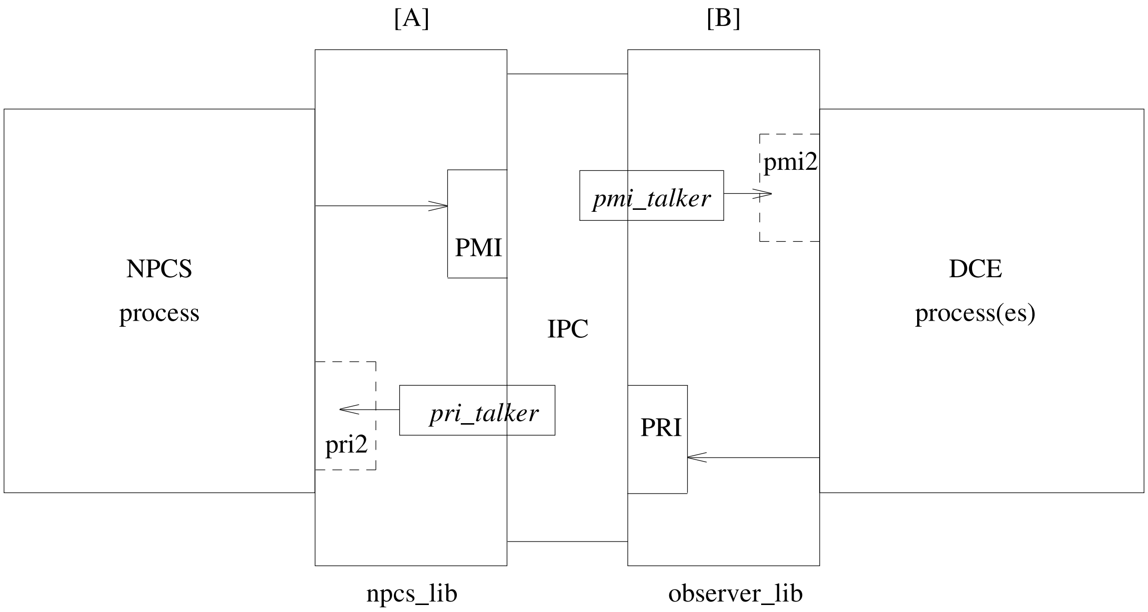

Figure 6. The architecture of the Encapsulated Library.

The model is illustrated in Figure 6. It provides two libraries which use/support the PRI and PMI interfaces. Servers of PRI and clients of PMI would link with `npcs_lib`. It is worthwhile to emphasize that there is only one NPCS server of PRI per node. This is denoted in the diagram as *pri2*, thus indicating the subset of PRI[B] (functions) specified through the PMI[A]. The point is that `npcs_lib` defines the functions (entry point symbols) named in the PMI specification and `observer_lib` defines the functions named in the PRI specification.

Servers of PMI and clients of PRI would link with `observer_lib`. This is very analogous to DCE RPC client and server stubs. The libraries may create threads needed to support asynchronous communication. The *pmi_talker* and *pri_talker* threads are shown in Figure 6, and are named ‘‘talker’’, to contrast with RPC listener threads. The middle region labeled *IPC* represents an intra-node IPC mechanism whose choice is unspecified as long as the PRI and PMI interfaces provide the connecting mechanisms described in this API section. This flexibility will permit many implementation approaches without requiring ANY modification to the NPCS or DCE processes. The interface is made independent of the underlying IPC mechanism, by the use of procedures provided by the recipient (server) of a request, which are invoked whenever a (client) request is made. This is analogous to an RPC, but to allow for a more general implementation the procedure names are passed to the libraries as procedure-valued parameters to the initialization calls: `dms_pmi_el_initialize` in section 10.2, and `dms_pri_el_initialize` in section 11.2.

The subset of the PRI functions passed to the PMI is denoted as *pri2* in Figure 6. These functions perform local initialization, and then take whatever steps are required to open a communication path for the processes to communicate. The exact nature of these steps depend on the particular implementation of the PMI/PRI interface. Possibilities include, but are not limited to:

(a) Creating a pair of named pipes (fifos).

(b) Calling `dciInitialize()/dciRegister()` (see the discussion regarding the DCI in [CMG]).

(c) Initializing a DCE RPC interface which accepts the needed procedures as RPCs.

(d) Creating a shared memory segment and initializing it with appropriate structures, the monitor threads to dequeue input messages, and semaphores to control access to the message queues.

(e) The encapsulated library requires several utility functions for library initialization and clean up. These are described in detail in sections 10 and 11, and summarized here. The `dms_pmi_el_initialize()` and `dms_pri_el_initialize()` functions are used to initialize the library and underlying IPC mechanisms. The `dms_pmi_el_free_outputs()` and the `dms_pri_el_free_outputs()` functions are used for freeing up memory resources, and encapsulate RPC free routines if necessary.

7.3. Important State Information

This section summarizes the important state maintained or passed via the standard interfaces.

7.3.1. Sensor data and reporting data structures

Sensor data components are described by `sensor_data` of type `dms_datum_t`. These types allow a wide range of sensor data representations, including opaque data structures for extensibility.

Sensor data is reported using the `sensor_report_list` of type `dms_observations_data_t`.

```
typedef struct dms_opaque {
    unsigned long        size;
    [size_is(size)] byte bytes[];
  } dms_opaque_t;

typedef enum {
    dms_LONG,
    dms_HYPER,
    dms_FLOAT,
    dms_DOUBLE,
    dms_BOOLEAN,
    dms_CHAR,
    dms_STRING,
    dms_BYTE,
    dms_OPAQUE,
    dms_DATA_STATUS
  } dms_datum_type_t;

typedef union dms_datum
    switch (dms_datum_type_t type) {
    case dms_LONG:
        long long_v;
    case dms_HYPER:
        hyper hyper_v;
    case dms_FLOAT:
        float float_v;
    case dms_DOUBLE:
        double double_v;
    case dms_BOOLEAN:
        boolean boolean_v;
    case dms_CHAR:
        char char_v;
    case dms_STRING:
        dms_string_t *string_p;
    case dms_BYTE:
        byte byte_v;
```

```
        case dms_OPAQUE:
            dms_opaque_t *opaque_p;
        case dms_DATA_STATUS:
            error_status_t status_v;
      } dms_datum_t;

    typedef struct dms_sensor_data {
        dms_sensor_id_t                sensor_id;
        unsigned long                  count;
        [size_is(count)] dms_datum_t   sensor_data[];
      } dms_sensor_data_t;

    typedef struct dms_timevalue {
        unsigned long sec;
        unsigned long usec;
      } dms_timevalue_t;

    typedef struct dms_observation_data {
        dms_timevalue_t                    end_timestamp;
        unsigned long                      count;
        [size_is(count)] dms_sensor_data_t* sensor[];
      } dms_observation_data_t;

    typedef struct dms_observations_data {
        unsigned long                           count;
        [size_is(count)] dms_observation_data_t* observation[];
      } dms_observations_data_t;
```

7.3.2. Sensor naming and registration data structures

Sensors are registered using the `sensor_register_list` of type `dms_instance_dir_t`.

Sensors in the sensor registry are named using the `registry_list` of type `dms_instance_dir_t`.

```
    /* This interface defines the data structures that represent
     * the dms namespace.  There are two forms of names that can be
     * represented, a simple string-only form, and a fully
     * decorated form.
     */

    typedef struct dms_name_node*  dms_name_node_p_t;

    typedef struct dms_name_nodes {
        unsigned long                      count;
        [size_is(count)] dms_name_node_p_t names[];
      } dms_name_nodes_t;

    typedef struct dms_name_node {
        dms_string_t*     name;  /*"*" == wildcard*/
        dms_name_nodes_t  children;
      } dms_name_node_t;

    typedef struct dms_attr {
        dms_string_t* attr_name;
        dms_datum_t   attr_value;
      } dms_attr_t;
```

```
typedef struct dms_attrs {
    unsigned long    count;
    [size_is(count)] dms_attr_t* attrs[];
  } dms_attrs_t;

typedef struct dms_sensor {
    dms_sensor_id_t  sensor_id;
    dms_attrs_t*     attributes;
    unsigned short   count;
    [size_is(count)] small metric_id[];
  } dms_sensor_t;

typedef struct dms_instance_leaf {
    unsigned long                  count;
    [size_is(count)]  dms_sensor_t* sensors[];
  } dms_instance_leaf_t;

typedef struct dms_instance_node*  dms_instance_node_p_t;

typedef struct dms_instance_dir {
    unsigned long                         count;
    [size_is(count)] dms_instance_node_p_t children[];
  } dms_instance_dir_t;

typedef enum {
    dms_DIRECTORY,  dms_LEAF,  dms_NAME_STATUS
  } dms_select_t;

typedef union dms_instance_data
   switch (dms_select_t data_type) {
   case dms_DIRECTORY:
     dms_instance_dir_t*  directory;
   case dms_LEAF:
     dms_instance_leaf_t* leaf;
   case dms_NAME_STATUS:
     error_status_t         status;
  } dms_instance_data_t;

typedef struct dms_instance_node {
    dms_string_t*        name;
    dms_datum_t*         alternate_name;
    dms_instance_data_t  data;
  } dms_instance_node_t;
```

The naming data structure is illustrated in Figure 7.

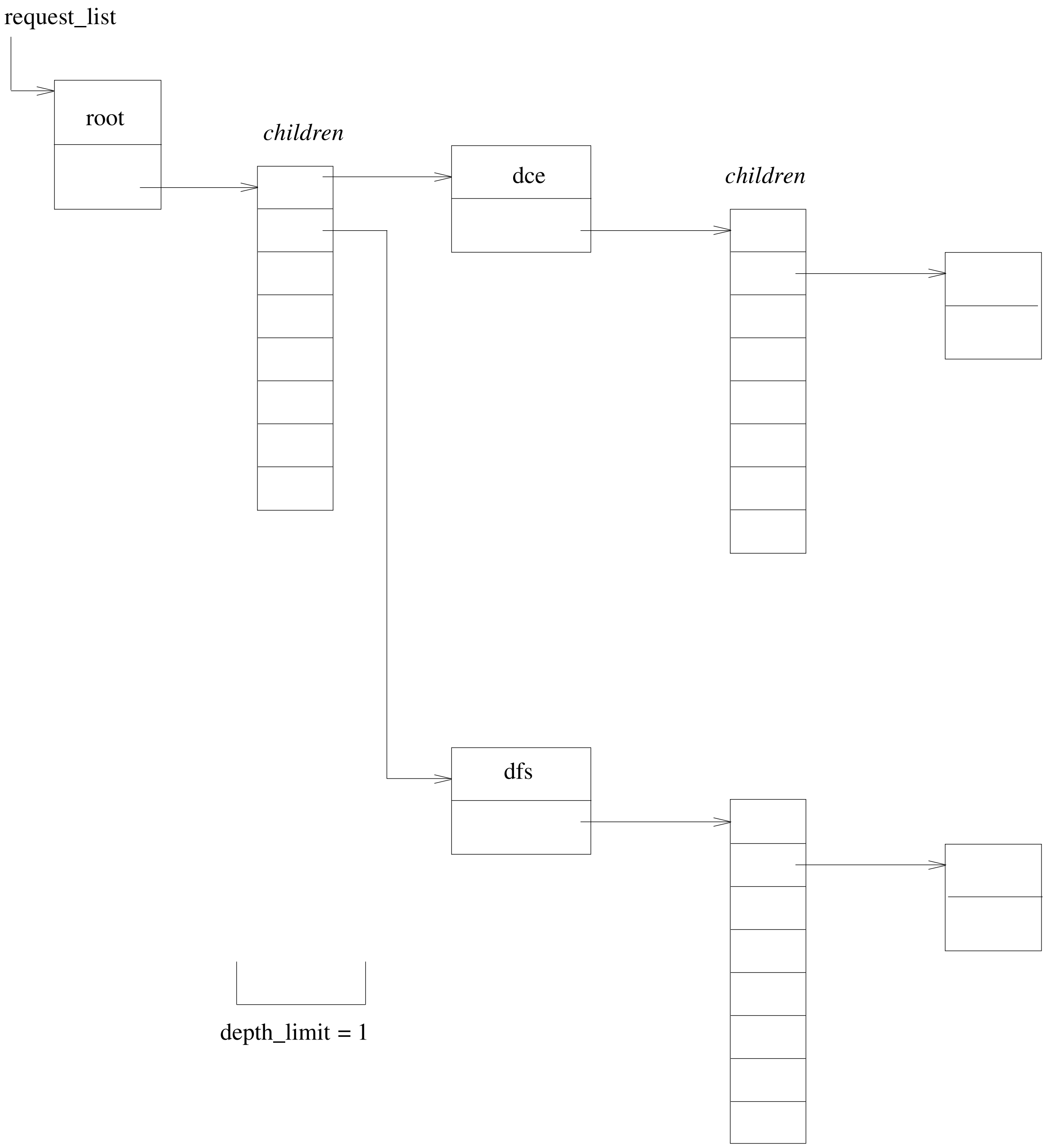

Figure 7. Sensor naming data structure. This example uses the parameters defined in the function `dms_npmi_get_registry()`, and shows the structures supporting the names `root/dce/...` and `root/dfs/...`, where ‘‘root’’ refers to the local network node where the NPCS resides. The depth parameter limits searches of subtrees.

7.3.3. Sensor configuration data structures

Sensor configuration data is returned in the `sensor_config_list` of type `dms_configs_t`.

```
const  unsigned long  dms_NO_METRIC_COLLECTION    = 0;
const  unsigned long  dms_THRESHOLD_CHECKING      = 0x00000001;
const  unsigned long  dms_COLLECT_MIN_MAX         = 0x00000002;
const  unsigned long  dms_COLLECT_TOTAL           = 0x00000004;
const  unsigned long  dms_COLLECT_COUNT           = 0x00000008;
const  unsigned long  dms_COLLECT_SUM_SQUARES     = 0x00000010;
const  unsigned long  dms_COLLECT_SUM_CUBES       = 0x00000020;
const  unsigned long  dms_COLLECT_SUM_X_TO_4TH    = 0x00000040;
const  unsigned long  dms_CUSTOM_INFO_SET         = 0x80000000;

typedef unsigned long dms_info_set_t;

typedef struct dms_threshold_values {
    dms_datum_t lower_value;
    dms_datum_t upper_value;
  } dms_threshold_values_t;

typedef union dms_threshold
    switch (boolean have_values) {
    case TRUE:
        dms_threshold_values_t values;
    case FALSE:
        ;
  } dms_threshold_t;

typedef struct dms_config {
    dms_sensor_id_t  sensor_id;
    dms_timevalue_t  reporting_interval;  /*0 == infinite*/
    dms_info_set_t   info_set;
    dms_threshold_t* threshold;
    error_status_t     status;
  } dms_config_t;

typedef struct dms_configs {
    unsigned long                    count;
    [size_is(count)] dms_config_t   config[];
  } dms_configs_t;
```

7.3.4. DMS binding data structures

Several ‘‘handles’’ are defined to bind elements, speed up searching and decrease communication costs.

(a) **Sensor ID** — To speed up searching for sensors in the NPCS registries, the specification defines a ‘‘handle’’, i.e., a shorthand, 32-bit reference that is unique per NPCS (and hence each node). This handle is called a sensor ID, and it is assigned by the NPCS at the time of initial sensor registration. This same handle is then provided to the PMA for its use.

(b) **Process index** — Shorthand provided by NPCS to speed observer/NPCS communication. Allows NPCS to search for all sensors for a particular process identifier (PID).

(c) **NPCS index** — Shorthand provided by PMA to speed PMA/NPCS communication. Allows PMA to rapidly identify sensor data reported by a particular NPCS.

(d) **PMA index** — Shorthand provided by NPCS to speed PMA/NPCS communication. Allows NPCS to rapidly identify requests of a particular PMA.

```
/* This interface defines the data structures used to represent
 * relationships between entities (sensors, processes, nodes)
 * within DMS.  Some are transparent, meaning that a user of
 * that structure can manipulate its contents.  Some are
 * opaque, meaning that only the creating entity can manipulate
 * its contents.
 */

/* TRANSPARENT BINDING TYPES */

typedef [string] unsigned char dms_string_t[];

typedef unsigned long  dms_protect_level_t; /* see rpc.h */

typedef [string] unsigned char  dms_string_binding_t[];

/* OPAQUE BINDING TYPES */

typedef unsigned long  dms_pma_index_t;

typedef unsigned long  dms_npcs_index_t;

typedef unsigned long  dms_process_index_t;

typedef unsigned long  dms_sensor_id_t;

typedef struct dms_sensor_ids {
                     unsigned long    count;
    [size_is(count)] dms_sensor_id_t  ids[];
  } dms_sensor_ids_t;
```

7.3.5. Sensor registry

The **sensor registry** contains descriptive information about the sensors located on a particular node. This registry is maintained by the NPCS. An entry contains:

(a) Sensor name (full sensor name in both string and OID format; this includes node, process, metric, and instance names).

(b) Sensor help text that describes the collected metric.

There is no explicit interface for obtaining modifications to the sensor registry. The PMA must periodically request interested sensors and compare this with previous requests.

7.3.6. Sensor configuration registry

A **configuration registry** contains configuration state about the sensors located on a particular node. This registry is maintained by the NPCS. An entry contains:

(a) Sensor name (full sensor name in both string and OID format; this includes node, process, metric, and instance names).

(b) Sensor information set.

(c) Sensor threshold values.

(d) Sensor summarization interval.

This may be combined with the sensor registry within the NPCS.

There is no explicit interface for obtaining modifications to the sensor configuration registry.

7.3.7. Sensor and metric attributes

There are several sensor and metric attributes. These include:

(a) Threshold.

(b) Units (e.g., kilobytes, seconds, etc.).

(c) Metric identifier.

(d) Metric name.

(e) Help text.

(f) Information sets supported.

(g) Sensor value subcomponent.

```
typedef enum {
        dms_METRIC_ID,
        dms_METRIC_DATUM_TYPE,
        dms_DATA_LENGTH,
        dms_METRIC_TYPE,
        dms_METRIC_NAME_INDEX,
        dms_HELP_TEXT_INDEX,
        dms_INFO_SET_SUPPORT,
        dms_SENSOR_UNITS,
        dms_LAST_ATTRIBUTE      /* this should remain last */
        } dms_attribute_t;
```

Runtime behavior for sensor value subcomponent attributes is described below:

(a) Minimum and maximum are RESET for each reporting interval.

(b) Counters and timers are accumulated continuously.

(c) Thresholds can support above, below, or a range of values to check against. Since the NPCS performs this test multiple thresholds values can be set for each sensor.

7.3.8. OSF global sensor registry

OSF must maintain a **global sensor registry** similar to the IETF SNMP registry [Rose], allowing vendors to provide globally known metrics and sensors but preserving local (vendor) autonomy and number assignment. This registry should be divided into domains analogous to the sensor naming described in section 5.1, to ease administration and interpretation of the sensors.

These ‘‘official’’ sensors are registered within the CDS when the DCE cell is brought up, and updates are registered as new versions of DCE are started within the cell.

A user branch must be available in the global sensor registry so that application developers may place well-known metrics and sensors there. An experimental branch should be supported to be used however deemed in each cell.

The specification proposes that this registry have the following tree structure (note that each entry level listed below represents a subdirectory; object identifiers are shown in parentheses following names):

(a) internet (1)

  (i) osf (5)

    [a] dce (1)

    [b] dfs (2)

    [c] security (3)

[d] cds (4)

[e] user (5)

[f] experimental (6)

[g] vendor (7)

[i] digital (1)

[ii] gradient (2)

[iii] hp (3)

[iv] hitachi (4)

[v] ibm (5)

[vi] informix (6)

[vii] microsoft (7)

[viii] novell (8)

[ix] oracle (9)

[x] sun (10)

[xi] transarc (11)

The above tree ignores the other branches already in use with the Internet SNMP community. We have added a branch for OSF with object identifier 5 (this value requires verification with IETF). Under the OSF branch are several subtrees for various DCE services. The user branch is unique to each customer's cell, and contains the results of custom sensors registered by user applications as described in section 7.4. The experimental subtree is for temporary use within a cell. The vendor subtree allows vendors the autonomy to assign and manage their custom sensors without requiring intervention from OSF. These vendor sensor must be registered within the cell in the same way as user custom sensors.

The OSF needs to work with the Internet Assigned Numbers Authority to register sensors and attributes.

7.4. Storing Custom Sensor Attributes in a Global Repository

Custom sensor attributes must be registered and stored in the CDS so that they are available to all PMAs in the cell. This specification recommends that they be stored in the CDS with the form:

`/.:/dms/sensors/`*`domain`*

where *`domain`* is one of `dce`, `dfs`, `security`, `cds`, `user`, `experimental` or `vendor`.

7.5. Security

It is a requirement to provide secure network transmission of performance data if mandated by local administrative policies. This allows protection against unauthorized users obtaining cleartext names of server processes, interfaces, operations or binding handles; falsifying client or server identities; or modifying transported data.

What are the implications on the 4 interfaces defined? The 2 control interfaces, NPMI and PMI, must be protected by access control to ensure that configuration data is modified only by those with proper authorization. The 2 data transport interfaces, PRI and NPRI, must be free from eavesdropping.

This specification assumes that intra-node communication via the PMI and PRI is secured by the host OS or the communication mechanism used. Consequently, it is not addressed further here.

To ensure that clients and servers are authentic, this specification recommends the creation of the new DCE security group, `perf_admin`, and enroll each host in this group. Principals for this group must be added to the security registry, and both the PMA and NPCS must login and execute as one of the principals (refreshing credentials programmatically as necessary). The host key is already available on the node and is automatically changed every 30 minutes. The benefit of making `perf_admin` a group is that the performance principal on each host (node) can change passwords independent of other hosts (nodes).

The NPCS must be able to execute as the owner of the performance principal's keytab file. Since the NPCS must be able to assume the identity of the host, it must run as root. However, this specification does not recommend that the NPCS run as root, but rather with a separate identity with sufficient capabilities to utilize DCE security services.

This does not solve the problem of users who can become root on a local host, and thereby become a member of the `perf_admin` group. Implementations of the measurements system should not preclude the extension of supporting several performance administration groups to address this security hole, when needed in ‘‘hostile’’ environments.

Authorization must be handled through the use of a reference monitor hard-coded into the manager routines of the NPMI and NPRI. The security policy enforced via this reference monitor is that clients with the `perf_admin` principal identity are authorized to invoke an NPMI or NPRI function. Client requests with any other principal identity should be rejected.

This reference monitor is universally enforced across all functions of the NPMI and NPRI. (It is possible to create an ACL manager that provides a much richer set of authorization capabilities, but that is beyond the scope of this version of the specification.) The reference monitor does not require support from IDL parameters, since the reference manager code obtains security information directly from the local RTL prior to processing the NPMI or NPRI function. (Note that the X/Open DCI uses a security key as a parameter. The PMI and PRI routines do not explicitly refer to this parameter, since that it is an implementation detail encapsulated by the PMI and PRI, and should be transparent to the calling process.)

Authenticated RPCs are used to address eavesdropping. Parameters in string form can appear for both NPMI and NPRI functions. The RPC data protection level is specified by the PMA when it first registers with the NPCS. Because all NPCSs may not support the same maximum protection level (for example, some data encryptions algorithms may not be available world-wide due to international export laws), the NPCS responds to the PMA request with the actual protection level that it can support. The PMA may unregister from this NPCS if the actual protection level is insufficient. The actual protection level can be set during sensor registration by specifying a minimum data protection level. This allows application developers and system managers to jointly specify the data protection level on an application basis if necessary. The policy enforced by the NPCS is the maximum of the PMA request and the sensor specified. The NPCS may also refuse service to a PMA that does not meet its minimum security requirements.

The use of a keytab file is also required (to hold the encryption key) for authenticated RPC, and implies that the NPCS executes with a dedicated user identifier to protect the keytab from unauthorized users. Although not recommended, unauthenticated RPC requests can be optionally supported by an NPCS on an implementation-dependent basis (this requires a configuration or command line parameter to enable).

The security policy outlined here does not prevent a PMA from accessing another's NPRI interface. Since this is an interface for trusted users (i.e., `perf_admin` principal), it is expected that PMA developers not invoke another NPRI.

PMAs that support cross-cell monitoring must use cross-cell authentication mechanisms prior to contacting an NPCS in a separate cell.

7.6. Error Conditions

Errors are described for each of the four APIs. Error conditions are returned in the `error_status_t` function return parameter. A general engineering philosophy is that error conditions should not be used to convey non-error-related state. This will assure efficient use of exception handling code for future implementations that decide to use C++. These function errors are described in detail in appendix I.

7.7. DMS Naming Convention

The following naming conventions are used in this specification:

(a) APIs are prefaced with the lower-case acronym of the distributed measurement systems concatenated with the interface name; e.g., `dms_pmi_`.

(b) API names use verbs and nouns separated by underscores; e.g., `dms_pmi_get_sensor_data()`.

(c) API names use the SNMP GET and SET verbs when applicable. This specification uses the verb REPORT for those interfaces that are push-based.

(d) Parameter names are in lower-case, separated by underscores. Names should make it clear whether a variable is a value or a pointer to a value, by using the suffix `_p` for pointers. Type names should end with the suffix `_t`. String names will end with the suffix `_str`.

7.8. API Description Format

The next four sections describe the standard APIs:

(a) The NPMI is described in section 8.

(b) The NPRI is described in section 9.

(c) The PMI is described in section 10.

(d) The PRI is described in section 11.

Each of the functions is described with the following format:

(a) The description provides a programmer's overview of the function's actions.

(b) The IDL provides the functions input and output parameters and types.

(c) The function input briefly describes each input parameter and its use (see section 7.3 for details on primary data structures).

(d) The function output briefly describes each input parameter and its use (see section 7.3 for details on primary data structures).

(e) The possible errors are summarized with a likely cause identified.

(f) The engineering notes provides explicit recommendations to the implementor (and not the user) of the function.

8. NPMI INTERFACE

The NPMI and NPRI interfaces are used by the PMAs to access and control sensors on any node in a DCE cell. The NPMI is supplied by the NPCS on each node. The NPRI is an optional, although recommended, interface provided by the PMA. The NPMI is described in this section, and the NPRI in section 9.

The NPMI interface provides each PMA with its own view of the sensors on a node in the DCE environment. Each PMA communicates with the NPCS to arrange delivery of sensor data via the NPMI or NPRI interfaces. The NPMI interface requires that PMA's explicitly discover and enable (configure) sensors, and then receive changed sensor data as it is pushed to them by the NPCS via the PMA's NPRI server interface. Specifically, the NPMI supports registering and unregistering PMAs interested in local sensors, getting and setting sensor configuration, and getting sensor data in a polled manner.

The NPMI is an RPC interface that is exported by the NPCS. Since this interface is accessed over the network, a non-RPC implementation is not recommended for security reasons. The NPMI functions pass parameters that local system administration policies may require protection from reading or modifying over a network. Therefore, the use of RPC data protection is supported for all NPMI functions (except for the initial act of registering a PMA).

The following Figure 8 illustrates the relationship between the physical sensors in an instrumented process and the PMA's logical view of sensors that is supported through the NPMI. Sensors are located in distinct processes and communicate with the NPCS via the observer. Each PMA, however, is only aware of the NPCS and sensors; the observer is transparent to the PMA.

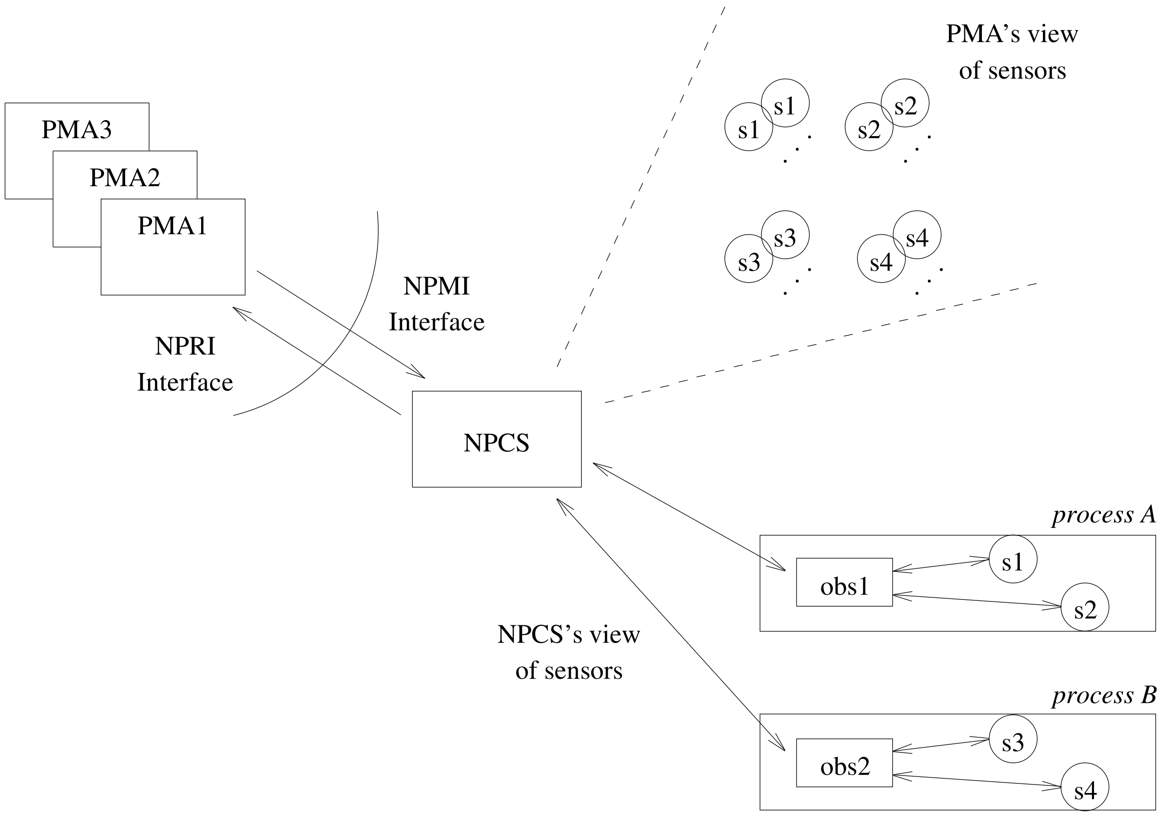

Figure 8. PMA versus NPCS view of sensors. A PMA's view of a sensor is limited to its own configuration request. The NPCS maintains the configuration state of all sensors on its node for all interested PMAs. In this example there are four sensors: *s1*, *s2*, *s3*, *s4*, and three PMAs: *PMA1*, *PMA2*, *PMA3*. For sensor *s1*, *PMA1* and *PMA3* have it enabled, while *PMA2* does not. Similarly, for sensor *s2*, *PMA1* does not have it enabled, while *PMA2* and *PMA3* do. The observer in each process (*obs1* and *obs2*) control requests and data from the NPCS and the sensors.

8.1. NPMI IDL

The complete IDL file is provided in appendix E.

8.2. dms_npmi_register_pma()

8.2.1. Description

This interface is provided by NPCS to allow PMAs to establish a connection. A PMA uses this interface to register its existence, the binding handle of its NPRI, and to establish data protection levels.

Any PMA that requests a greater protection level than specified by the **minimum_protection_level** will have to decide whether to continue. The protection level will be applied to parameters of all function calls and to ALL sensor data transported from this node to the PMA via the NPRI. This may cause excessive overhead, so it should be used with caution.

If a new instrumented process begins execution and requires a higher protection level than that in place when a PMA previously registered with the NPCS, then the NPCS must not make any of this sensor data available to the PMA until the PMA re-registers with the proper protection level.

8.2.2. Function signature

```
error_status_t  dms_npmi_register_pma (
   [in    ] handle_t                handle,
   [in,ptr] dms_string_binding_t*  npri_binding,
                                    /*null == client-only PMA*/
   [in    ] dms_npcs_index_t        npcs_index,
   [in    ] dms_protect_level_t     requested_protect,
   [   out] dms_pma_index_t*        pma_index,
   [   out] dms_protect_level_t*    granted_protect
  );
```

8.2.3. Function input

(a) `handle` — RPC binding handle of NPMI.

(b) `npri_binding` — Pointer to a string binding handle of PMA's NPRI interface. If this is NULL, then the PMA does not support a NPRI.

(c) `npcs_index` — Unique identifier assigned by PMA that provides a shorthand for future NPCS-to-PMA communication.

(d) `requested_protect` — The PMA's requested level of RPC data protection for use in subsequent NPMI calls, or when data is returned via NPRI functions.

8.2.4. Function output

(a) `pma_index` — Unique identifier assigned by NPCS that provides a shorthand for future PMA-to-NPCS communication.

(b) `granted_protect` — The NPCS's granted level of RPC data protection used by the NPCS when returning data via NPRI functions, or for subsequent NPMI functions. It might not be the same as that requested by the PMA. It is established by the system manager at NPCS execution time.

(c) *Function return value* — `dms_status` — status of call; non-zero if call encountered an error.

8.2.5. Errors

(a) `REGISTER_FAILED` — NPCS unable to complete registration.

(b) `ALREADY_REGISTERED` — PMA previously registered.

(c) `PROTECT_LEVEL_NOT_SUPPORTED` — Requested data protection level not supported; `granted_protect` will be used.

(d) `ILLEGAL_BINDING` — Binding handle illegal.

8.2.6. Engineering notes

(a) Datagram RPC communication to the NPRI interface is recommended. This eliminates the overhead of TCP/IP connection/teardown for infrequent communication. The rest of the infrastructure has been designed to minimize the effects of lost packets, should they occur.

(b) The PMA client code must inform its NPRI server of the granted data protection level used by the NPCS for subsequent NPRI invocations. The reference monitor of the NPRI controls whether requests with the granted data protection level specified by the NPCS are acceptable, based on its supported minimum protection level.

(c) It is not necessary for the NPCS to register the NPMI in the CDS. Instead, the UUID can be converted to a string, concatenated with the NPCS node IP address, and a call made that the `dced` will deliver. The UUID of the NPMI is specified in section 8.1.

(d) The use of context handles between the PMA and NPCS are not recommended, because some PMAs will be client-only or single-threaded, and the amount of ‘‘still alive’’ traffic between the RTLs must be minimized. The failure modes and recovery actions described in section 13.8 should be implemented instead of context handles.

(e) It is possible for a PMA to register multiple times with the same NPCS. This allows the PMA to support different NPRI interfaces in the same or different processes. The NPCS should return a unique `pma_index` for each of these registrations.

(f) This function supports non-idempotent semantics.

8.3. dms_npmi_get_registry()

8.3.1. Description

This interface is provided by NPCS to supply all or part of the node-level sensor registry. The PMAs use this interface to discover available sensors and the current configuration state.

The NPCS does not support a synchronous event function to notify a PMA of changes to the sensor registry, namely the addition and deletion of individual sensors. The PMA must periodically invoke this function with interested sensors in the `request_list` and compare the results with previous calls to determine what changes have occurred within the sensor registry. This function should be used sparingly for this need, to minimize network resource utilization.

The registry structure is defined by the data structures in section 7.3.2, and is illustrated in Figure 7.

There is no support for a ‘‘wildcard’’ using regular expressions. Rather, the tree of interest is provided in the `request_list` with a `depth_limit`, and all subtrees matching these constraints are returned. This bulk-input parameter allows support for requesting multiple sensors in a single call to this function. However, a more generalized query processor is delegated to the PMA, which must then translate requests to this function.

If the requested `depth_limit` is greater then the implicit `depth_limit` of the `request_list`, then this function returns the sensors at a depth equal to that of the `request_list`. Otherwise, only the requested `depth_limit` of the registry is returned.

Requests can only be made with string sensor instance names.

8.3.2. Function signature

```
error_status_t  dms_npmi_get_registry (
   [in     ]  handle_t               handle,
   [in     ]  dms_pma_index_t        pma_index,
   [in,ptr]   dms_name_nodes_t*      request_list,
                                     /*null == entire registry*/
   [in     ]  long                   depth_limit,
                                     /*0 == infinity*/
   [    out]  dms_instance_dir_t**   registry_list
  );
```

8.3.3. Function input

(a) `handle` — RPC binding handle of NPMI.

(b) `pma_index` — Unique identifier assigned by NPCS that provides a shorthand for NPCS-to-PMA communication. This also provides a test to determine whether NPCS has terminated and restarted since the last `dms_npmi_get_registry()` call, because a new NPCS won’t know this value.

(c) `request_list` — A pointer to a tree of sensor names that the PMA is interested in. This parameter uses a tree structure that contains one or more subtrees. If the pointer is NULL, then the entire registry is returned.

(d) `depth_limit` — This limits the search depth, and consequently the number of subtrees, returned by the NPCS. This value is the number of nodes starting with the ‘‘root’’ node of the NPCS sensor registry. If this value is 0, then all subtrees are returned.

8.3.4. Function output

(a) `registry_list` — Registry data for one or more sensors that satisfy the `request_list`. The sensor identifiers contained within this structure are used by the PMA for subsequent configuration actions, and to identify sensor data reported via the NPRI.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

8.3.5. Errors

(a) `UNKNOWN_PMA` — PMA not registered.

(b) `UNKNOWN_SENSOR` — One or more sensors included in `sensor_list` were not registered.

8.3.6. Engineering notes

(a) Since RPC parameters include sensor names, this interface must have the option of supporting RPC data protection. This is accomplished via the `dms_npmi_register_pma()` call.

(b) If a DCE service-oriented view of a process name is used (e.g., `/.:/sec`), then the PMA must translate this to a legal sensor name before contacting the NPCS.

(c) This function supports idempotent semantics.

8.4. dms_npmi_set_sensor_config()

8.4.1. Description

This interface is provided by NPCS to allow PMAs to configure which sensor metric components to collect, and reporting frequency. This view of the sensor is unique to each requesting PMA (`pma_index`), and conflicts, if any, are arbitrated by the NPCS. Requested configuration changes are set on a sensor-by-sensor basis.

A list of `sensor_configs` is used to request configuration, and to return configuration status. Only sensors that could not be set to requested configuration state are returned, along with their current configuration state. If a sensor cannot be set to one or more of the requested parameters, then no configuration changes are made to the sensor. No sensor data will be reported for sensors that were not successfully configured. PMA must re-invoke this function with acceptable configuration parameters before data will be returned for a sensor.

The PMA also uses this function to disable sensors it is no longer interested in collecting data on. It does this by providing a list of sensors in `sensor_configs` with the `info_set` value set to 0.

There is no explicit support in this specification for getting sensor configuration data since this function can satisfy this need.

8.4.2. Function signature

```
error_status_t  dms_npmi_set_sensor_config (
   [in    ]  handle_t          handle,
   [in    ]  dms_pma_index_t   pma_index,
   [in,out]  dms_configs_t**   sensor_configs
  );
```

8.4.3. Function input

(a) `handle` — RPC binding handle of NPMI.

(b) `pma_index` — Unique identifier assigned by NPCS that provides a shorthand for NPCS-to-PMA communication.

(c) `sensor_configs` — A list of sensor identifiers and configuration state that PMA is interested in.

8.4.4. Function output

(a) `sensor_configs` — A list of sensor identifiers, status of configuration request, and configuration state returned by the NPCS. Only sensors that could not be configured as requested are returned in this structure.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

8.4.5. Errors

(a) `UNKNOWN_PMA` — PMA not registered.

(b) `UNKNOWN_SENSOR` — One or more sensors included in `sensor_list` were not registered.

(c) `NO_SENSOR_REQUESTED` — `sensor_configs` contains no sensors.

(d) `FUNCTION_FAILED` — The set operation failed due to one or more specified parameters conflicting with a previous request. No sensor configuration modifications were made.

(e) `UNKNOWN_INFO_SET` — Information set level out of range.

(f) `UNKNOWN_THRESHOLD_LEVEL` — Threshold level out of range.

8.4.6. Engineering notes

(a) The NPCS must arbitrate conflicting PMA requests for reporting interval, sensor information sets, and sensor threshold values, as described in section 12.2 on NPCS functions.

(b) No partial sensor configuration changes are supported. If a sensor cannot be set to all requested configuration values, then NONE of them will be set (i.e., leave sensor state unchanged).

(c) This function does not support idempotent semantics, since sensor registry changes may occur during a requested set operation.

8.5. dms_npmi_get_sensor_data()

8.5.1. Description

This interface is provided by NPCS to permit a poll of metric data without waiting for the next reporting interval. The sensor data is returned as an [out] parameter of the RPC.

Users of this interface include SNMP agents, PMAs with a monitoring policy of an occasional ‘‘one-shot’’ request, client-only PMAs, and special monitors for benchmarking or load-balancing that capture state before and after a workload’s execution.

To access the current content of a sensor set the `bypass_cache` flag to TRUE. This forces the NPCS to collect requested sensor data by invoking `dms_pmi_get_sensor_data()` for each requested process. This provides current sensor data, but is very costly. When the flag is FALSE, the NPCS returns the latest complete version of sensor data from its internal cache. The NPCS never returns data from a ‘‘partial interval’’, only the latest complete interval. This is much more efficient, but may provide ‘‘old’’ sensor data, depending on the sensor reporting interval.

If the ‘‘bypass_cache’’ flag is TRUE, then this function has the side-effects of resetting all sensor minimum and maximum values. This is because the action of a poll, by definition, results in the termination of the current summarization interval. The observer’s next scheduled reporting interval, if there is one, is not affected. To prevent these side-effects from affecting other PMAs that receive this data, a PMA using this function must first set the sensor reporting interval to `NO_REPORT_INTERVAL`. This interval value is also used by the NPCS to ensure that only one PMA in the cell can access this sensor using this function, since this mode assumes that only one PMA ‘‘owns’’ the sensor and wants no interference from other PMA requests. All other PMAs are then prevented from modifying the sensors configuration, although they can access its data. These side-effects do not occur if the ‘‘bypass_cache’’ flag is FALSE.

This get operation will fail if the PMA has not previously registered and set the sensor configuration correctly. In this failing case, a NULL list of `sensor_data` is returned.

The use of this ‘‘polling’’ interface is discouraged, since it requires significant network bandwidth.

8.5.2. Function signature

```
error_status_t  dms_npmi_get_sensor_data (
   [in    ]  handle_t                   handle,
   [in    ]  dms_pma_index_t            pma_index,
   [in    ]  dms_sensor_ids_t*          sensor_id_list,
   [in    ]  boolean                    bypass_cache,
   [   out]  dms_observations_data_t**  sensor_data
  );
```

8.5.3. Function input

(a) `handle` — RPC binding handle of NPMI.

(b) `pma_index` — Unique identifier assigned by NPCS that provides a shorthand for NPCS-to-PMA communication; this handle is NULL for client only PMAs.

(c) `sensor_id_list` — A list of sensor identifiers that the PMA is interested in.

(d) `bypass_cache` — A flag that when TRUE forces the NPCS to collect requested sensor data directly from each sensor. This provides current sensor data, but is very costly. When the flag is FALSE, the NPCS returns the latest version of sensor data from the NPCS internal cache. This is much more efficient, but may provide ''old'' sensor data depending on the sensor reporting interval.

8.5.4. Function output

(a) `sensor_data` — One or more sensor identifiers and corresponding data are returned.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error

8.5.5. Errors

(a) `UNKNOWN_PMA` — PMA not registered.

(b) `UNKNOWN_SENSOR` — One or more sensors included in `sensor_list` were not registered.

(c) `NO_SENSOR_REQUESTED` — `sensor_list` contained no sensors.

(d) `BYPASS_NOT_ALLOWED` — Sensor configuration does not allow cache bypass, due to conflict with another PMA.

8.5.6. Engineering notes

(a) Since RPC parameters include sensor data, this interface must have the option of supporting RPC data protection. This RPC data protection level was set via the `dms_npmi_register_pma()` call.

(b) This interfaces does not support idempotent semantics.

8.6. dms_npmi_unregister_pma()

8.6.1. Description

This interface is provided by a NPCS to break the connection between a PMA and a NPCS, and free up NPCS resources. All sensors that have been configured by this PMA are disabled if the NPCS arbitration rules permit. PMAs use this interface to permanently break a connection. There is no support in this specification for a PMA temporarily suspending a connection.

Client-only PMAs (COPs) must use this interface to minimize resources unnecessarily consumed by the NPCS. The NPCS will maintain COP requests for a maximum interval of one between COP requests for getting sensor data.

8.6.2. Function signature

```
error_status_t  dms_npmi_unregister_pma (
   [in    ]  handle_t          handle,
   [in    ]  dms_pma_index_t   pma_index
  );
```

8.6.3. Function input

(a) `handle` — RPC binding handle of NPMI.

(b) `pma_index` — Unique identifier assigned by NPCS that provides a shorthand for NPCS-to-PMA communication.

8.6.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

8.6.5. Errors

(a) `UNKNOWN_PMA` — PMA not registered.

8.6.6. Engineering notes

(a) Must use the `granted_protect` returned in `dms_npmi_register_pma()` call — this may cause problems for international users whose PMAs and NPCS are in different countries with different export controls on the use of authenticated RPC. This issue is beyond the scope of this RFC.

(b) All unregister requests result in the NPCS freeing up resources and re-setting sensors to a quiescent state wherever that does not conflict with other PMA requests.

(c) The NPCS should conduct a sanity check on the RPC binding handle (using string binding conversion), to disallow PMA1 from unregistering an NPCS request of PMA2.

(d) This interface does not support idempotent semantics.

9. NPRI INTERFACE

The NPRI's primary purpose is to provide a data transport channel so that a PMA can receive sensor data from an NPCS without the need to poll for each update. Specifically, this interface supports network reporting of a node's sensor data. All PMAs must implement this interface to receive data from an NPCS without the need to poll for it. However, a polling interface, `dms_npmi_get_sensor_data()`, is provided by the NPMI for simple or client-only PMAs (COPs). All other state information about NPCS and sensors is obtained explicitly by invoking the NPMI routines. To simplify the design the NPCS does not notify the PMA of changes in sensor or NPCS state.

The NPRI is an RPC interface that is a part of the PMA. Since this interface is accessed over the network a non-RPC implementation is not recommended, due to security issues. The PMA sets the data protection level of this interface in the `dms_npmi_register_pma()` call.

9.1. NPRI IDL

The complete IDL is located in appendix F.

9.2. dms_npri_report_sensor_data()

9.2.1. Description

This interface is provided by the PMAs to assimilate updated sensor metric components without the need for polling. All sensor data that has changed within the last reporting interval is packaged together by the NPCS and reported in a single report.

The state diagram in Figure 9 illustrates when data is pushed from the NPCS to the PMA. All state transitions occur only at PMA-specified reporting interval boundaries, with the exception of the reconfiguration state transition, which occurs asynchronously with respect to reporting intervals. The nesting of state indicates a separate state machine for each PMA's view of a sensor configured.

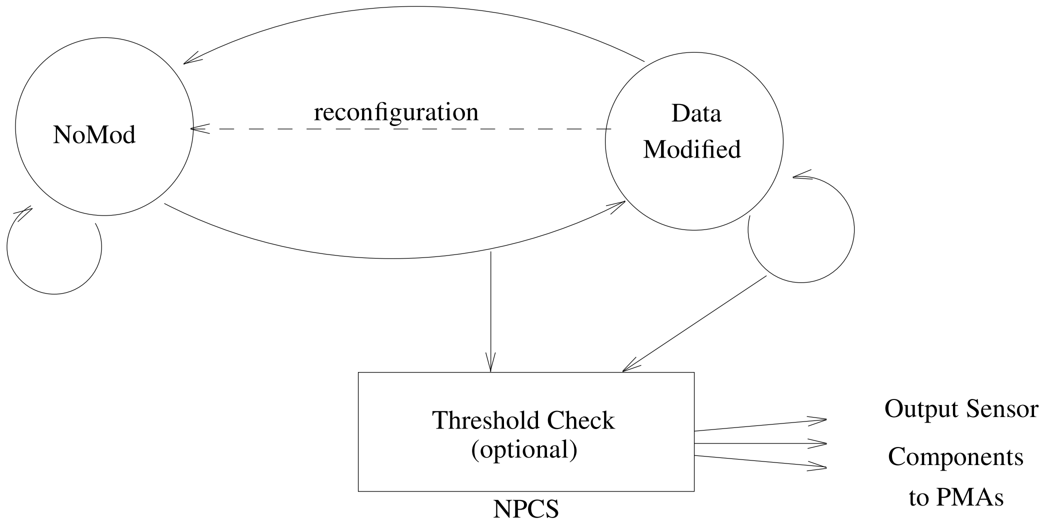

Figure 9. `dms_npri_report_sensor_data()` sensor state machine. Sensor data is pushed to the PMA by the NPCS only if it was modified during the current reporting interval.

This call requires the PMA to have previously registered with the NPCS, and provided a binding to its NPRI interface. Data will not flow to the NPRI until the PMA enables sensors using the `dms_npmi_set_sensor_config()` function.

The NPCS will return a NULL sensor data list of there is no sensor data to report for this interval. This serves as a ''still-alive'' message to the PMA during periods of application (and hence sensor) inactivity or when no thresholds were exceeded.

9.2.2. Function signature

```
error_status_t  dms_npri_report_sensor_data (
    [in    ] handle_t                  handle,
    [in    ] dms_npcs_index_t          npcs_index,
    [in,ptr] dms_observations_data_t*  sensor_data
                                       /*null == keep-alive*/
  );
```

9.2.3. Function input

(a) `handle` — The RPC binding handle of the NPRI.

(b) `npcs_index` — Unique identifier assigned by PMA by function `dms_npmi_register_pma()` that provides a shorthand for NPCS-to-PMA communication.

(c) `sensor_data` — A structure containing one or more sensors and the data components as configured by this PMA. See section 7.3.1 for details. May be NULL if no sensor data to report in this interval.

9.2.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

9.2.5. Errors

(a) `UNKNOWN_SENSOR` — Reported sensor not requested by PMA; PMA should call `dms_npmi_set_sensor_config()` and disable this sensor.

(b) `UNKNOWN_NPCS` — Reporting NPCS not recognized; PMA should re-register with this NPCS to reestablish a valid `npcs_index`.

9.2.6. Engineering notes

(a) NPRI routine not called for PMAs that register a NULL `npri_binding` handle in `dms_npmi_register_pma`.

(b) Our philosophy is to minimize the data sent across the network; consequently, the NPCS maintains a directory of sensor configurations by PMAs, and only sends requested sensors with requested configurations.

(c) This call returns a synchronous output, so that the NPCS can determine if the PMA is still executing. If the call times out, then the NPCS should restart, based on section 13.8.

(d) For efficiency use idempotent RPC semantics for this call.

10. PMI INTERFACE

The PMI and PRI are the two low-level interfaces. These interfaces are used by the observer and NPCS to control sensors and transmit state. These interfaces are provided by the DCE vendor and are transparent to the PMA developer.

The PMI's primary purpose is to provide a control and access interface to sensors located within a process that supports DCE instrumented services. An NPCS uses the PMI routines to set sensor configuration state, get sensor data state, and initialize and terminate the connection to the NPCS.

The PMI is implemented in the encapsulated library as described in section 7.2. The actual communication is implemented as either an RPC interface or as an implementation-specific IPC mechanism. The encapsulated library hides the actual communication mechanism from the programmer.

10.1. PMI IDL

The complete IDL is located in appendix G.

10.2. dms_pmi_el_initialize()

10.2.1. Description

This utility function is necessary to initialize the encapsulated library. It records the PRI procedures in private variables, and takes whatever steps are required to open a communication path for processes to communicate with NPCS. The exact nature of these steps depend on the particular implementation of the PMI/PRI interface. Possibilities include, but are not limited to:

(a) Creating a FIFO of known name and opening it for reading.

(b) Calling `dciInitialize()`.

(c) Initializing a DCE RPC interface, and creating a talker thread that accepts the PRI procedures as RPCs.

(d) Creating a shared memory segment and initializing it with appropriate structures, the PRI talker thread to dequeue input messages from instrumented processes, and a semaphore to control access to the queue.

10.2.2. Function signature

```
error_status_t  dms_pmi_el_initialize (
   [in    ] dms_pri_reg_proc_fp_t     pri_register_process,
   [in    ] dms_pri_reg_sensor_fp_t   pri_register_sensor,
   [in    ] dms_pri_report_data_fp_t  pri_report_sensor_data,
   [in    ] dms_pri_unreg_sensor_fp_t pri_unregister_sensor,
   [in    ] dms_pri_unreg_proc_fp_t   pri_unregister_process
  );
```

10.2.3. Function input

(a) `pri_register_process`

(b) `pri_register_sensor`

(c) `pri_report_sensor_data`

(d) `pri_unregister_sensor`

(e) `pri_unregister_process`

These are all callback (local) procedures exported by NPCS, invoked by the encapsulated library whenever the corresponding PRI procedure is invoked by an instrumented process. These procedures have identical signatures to their corresponding PRI procedures.

10.2.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

10.2.5. Errors

(a) `FUNCTION_FAILED` — Initialization function failed due to an internal encapsulated library error.

10.2.6. Engineering notes

(a) The details necessary to support the specific IPC mechanism are implementation-dependent, and transparent to this function.

10.3. dms_pmi_el_free_outputs()

10.3.1. Description

This utility function is necessary to initialize free output data in the encapsulated library, encapsulate RPC free memory functions, and eliminate possible memory leaks.

10.3.2. Function signature

```
error_status_t  dms_pmi_el_free_outputs (
   [in,ptr] dms_configs_t*          sensor_config_list,
                                    /*null == absent*/
   [in,ptr] dms_observation_data_t* sensor_report_list
                                    /*null == absent*/
  );
```

10.3.3. Function input

(a) `sensor_config_list` — A pointer to the sensor configuration list that the programmer desires to free allocated memory. Set this to NULL if no list is to be freed.

(b) `sensor_report_list` — A pointer to the sensor reporting list that the programmer desires to free allocated memory. Set this to NULL if no list is to be freed.

10.3.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

10.3.5. Errors

(a) `FUNCTION_FAILED` — Initialization function failed due to an internal encapsulated library error.

10.3.6. Engineering notes

(a) The details necessary to support the specific free memory mechanisms are implementation-dependent, and transparent to this function.

10.4. dms_pmi_terminate()

10.4.1. Description

This function disconnects the NPCS from all registered observers, and is useful for planned shutdowns of the NPCS. The function undoes the actions of the `dms_pmi_el_initialize()` function. The specific actions are implementation-dependent. The observer's response to this request is to return all sensors to a quiescent state.

There is no comparable call from the NPMI, so a PMA cannot cause this action. This call should be supported via the normal DCE control programs (such as `dcecp`).

10.4.2. Function signature

```
error_status_t  dms_pmi_terminate (
   void
  );
```

10.4.3. Function input

None.

10.4.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

10.4.5. Errors

(a) `FUNCTION_FAILED` — Terminate action failed due to an internal encapsulated library error.

10.4.6. Engineering notes

(a) The implementation specific encapsulated library must provide a mechanism that ensures that observer's calling any PRI function prior to receiving the `dms_pmi_terminate()` call can determine that the NPCS has stopped execution and should invoke its internal clean-up routines.

10.5. dms_pmi_set_sensor_config()

10.5.1. Description

This interface is provided to select which metric components (information set, etc.) a sensor supplies, and the interval between sensor summarizing and reporting those components. The NPCS uses this interface to set sensors on a per-process basis (i.e., for one observer at a time). Consequently, to set the sensors in *N* processes requires *N* invocations of this function (one call each to *N* observers).

All requested operations are done on a sensor-by-sensor basis only, for sensors requested in the `sensor_config_list`. No global sensor configurations are supported.

This function does not return verification status about each sensor configured. It returns status only on sensors that were not modified. Sensors are never left in a ''partially modified'' state. If any of the requested configuration states were not modified, then no sensor state is modified, and this current state is returned as function output with the appropriate error status. If ''all or nothing'' semantics are required, then the application must explicitly reset all sensors that were successfully set.

10.5.2. Function signature

```
error_status_t  dms_pmi_set_sensor_config (
   [in    ] dms_process_index_t  process_index,
   [in,out] dms_configs_t**      sensor_config_list
  );
```

10.5.3. Function input

(a) `process_index` — Shorthand provided by NPCS via `dms_pri_register_process()`.

(b) `sensor_config_list` — A list of sensor identifiers and requested configuration states.

10.5.4. Function output

(a) `sensor_config_list` — A list of sensor identifiers and resulting configuration states for sensors that could NOT be set to the requested level.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

10.5.5. Errors

(a) CHECK_INTERNAL_STATUS — Sensor configuration not changed due to non-existent sensor, illegal request, or previous state is mutually exclusive of requested state. The status of each failing sensor request is returned in the internal status fields of the sensor_config_list.

(b) Individual sensor errors are summarized in section 7.6.

10.5.6. Engineering notes

(a) The process_index is an input parameter for use by the encapsulated library to identify the requested observer.

(b) This function input can support setting sensors to a threshold level, even though this version of the specification requires this is a function of the NPCS for standard sensors. However, custom sensors might support thresholds that the NPCS cannot. Consequently, if no threshold is settable on the sensor of interest, then return NO_THOLD as an error.

10.6. dms_pmi_get_sensor_data()

10.6.1. Description

This function is provided as a ‘‘polling’’ interface that obtains current sensor data as function output. The function returns data for each sensor requested, whether the sensor data has changed in the last interval or not. A timestamp is also returned so that this data can be correlated with other measurements in the cell. This function is not directly callable by a PMA, but is only invoked when the dms_npmi_get_sensor_data() function is invoked with the bypass_cache flag set to TRUE.

This function has the side-effects of resetting all sensor minimum and maximum values. The observer’s next scheduled reporting interval, if there is one, is not affected. To prevent these side-effects from affecting other PMAs that receive their data in the recommended way, a PMA using this function must first set the sensor reporting interval to NO_REPORT_INTERVAL. This interval value is also used by the NPCS to ensure that only one PMA in the cell can access this sensor using this function.

This function is not the recommended method of obtaining sensor data, but is provided for compatibility with existing management applications (such as SNMP), and to support client-only PMAs. The recommended mode of access is using the PRI dms_pri_report_sensor_data() function, which is more efficient and scalable.

10.6.2. Function signature

```
error_status_t  dms_pmi_get_sensor_data (
   [in    ] dms_process_index_t      process_index,
   [in    ] dms_sensor_ids_t*        sensor_id_list,
   [   out] dms_observation_data_t** sensor_report_list
  );
```

10.6.3. Function input

(a) process_index — Shorthand provided by NPCS via dms_pri_register_process()

(b) sensor_id_list — A list of sensor identifiers as assigned by the NPCS via the dms_pri_register_sensor() function.

10.6.4. Function output

(a) sensor_report_list — Returns a list of sensors and individual values, and a timestamp that corresponds to when the observer returned the data.

(b) *Function return value* — dms_status — Status of call; non-zero if call encountered an error.

10.6.5. Errors

(a) UNKNOWN_SENSOR — Sensor does not exist, or unknown sensor identifier.

(b) SENSOR_NOT_CONFIGURED — Sensor not configured to collect data.

(c) SENSOR_CONFIG_CONFLICT — Sensor not configured for access via this method, since its reporting interval was not set to NO_REPORT_INTERVAL.

10.6.6. Engineering notes

(a) This function returns data in the same format as supplied by dms_pri_report_sensor_data(). The IPC mechanism for non-RPC implementations of the encapsulated library is implementation-dependent but must support this function's input and output parameters.

(b) This function output does not include the sensor data component containing the metric threshold value for this reporting interval, since that is a property of the NPCS for standard sensors.

(c) The timestamp returned in the sensor_report_list is obtained by the observer at the end of the reporting interval, i.e., after it has prepared sensor data for transport but just prior to actually transporting the data.

11. PRI INTERFACE

The PRI's primary purpose is to provide an efficient, interprocess data transportation channel for observer-to-NPCS communication. Specifically, the PRI supports routines to register processes (observers) and sensors, transmit (push) sensor data between the instrumented process's address space and the NPCS's, and unregister processes (observers) and sensors. The observer is the only DMS element allowed to invoke these routines. The registration routine is invoked prior to providing any data collection or support of PMI routines.

The PRI is implemented as either an RPC server interface exported by the NPCS, or as an IPC mechanism.

11.1. PRI IDL

The complete IDL is located in appendix H.

11.2. dms_pri_el_initialize()

11.2.1. Description

This utility function is necessary to initialize the encapsulated library. It records the PMI procedures in private variables, and takes whatever steps are required to locate the communication path to communicate with the instrumented process. The exact nature of these steps depend on the particular implementation of the PMI/PRI interface. Possibilities include, but are not limited to:

(a) Opening a FIFO of known name for writing.

(b) Calling dciRegister (see this function description in [CMG]).

(c) Obtaining a binding to the NPCS DCE RPC interface which accepts the PRI procedures as RPCs.

(d) Attaching to the shared memory segment created by NPCS, and creating the PMI talker thread to monitor an input queue of messages from NPCS.

11.2.2. Function signature

```
error_status_t  dms_pri_el_intialize (
   [in    ] dms_pmi_set_config_fp_t  pmi_set_sensor_config,
   [in    ] dms_pmi_get_data_fp_t    pmi_get_sensor_data,
   [in    ] dms_pmi_terminate_fp_t   pmi_terminate
  );
```

11.2.3. Function input

(a) pmi_set_sensor_config

(b) pmi_get_sensor_data

(c) pmi_terminate

These are all callback (local) procedures provided by the instrumented process, that are invoked by the encapsulated library whenever the corresponding PMI procedure is invoked by the NPCS. These procedures have identical signatures to their corresponding PMI procedures.

11.2.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.2.5. Errors

(a) `FUNCTION_FAILED` — Initialization function failed due to an internal encapsulated library error.

11.2.6. Engineering notes

(a) The details necessary to support the specific IPC mechanism are implementation-dependent, and transparent to this function.

11.3. dms_pri_el_free_outputs()

11.3.1. Description

This utility function is necessary to initialize free output data in the encapsulated library, encapsulate RPC free memory functions, and eliminate possible memory leaks.

11.3.2. Function signature

```
error_status_t  dms_pri_el_free_outputs (
   [in,ptr] dms_instance_dir_t* sensor_register_list
                                /*null == absent*/
  );
```

11.3.3. Function input

(a) `sensor_register_list` — A pointer to the sensor registration list that the programmer desires to free allocated memory.

11.3.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.3.5. Errors

(a) `FUNCTION_FAILED` — Initialization function failed due to an internal encapsulated library error.

11.3.6. Engineering notes

(a) The details necessary to support the specific free memory mechanisms are implementation-dependent, and transparent to this function.

11.4. dms_pri_register_process()

11.4.1. Description

This interface is invoked by instrumented DCE processes to provide the NPCS with the data necessary to build and maintain the node-level sensor registry. The observer in a DCE process uses this interface to register process specific state.

11.4.2. Function signature

```
error_status_t  dms_pri_register_process (
   [in    ] dms_string_t*        process_name,
   [in    ] long                 process_pid,
   [   out] dms_process_index_t* process_index
  );
```

11.4.3. Function input

(a) `process_name` — A string that contains the `argv[0]` value of the instrumented DCE process.

(b) `process_pid` — The value returned by `getpid()`.

Note that these function inputs are described for an operating system exporting a POSIX-conformant interface.

11.4.4. Function output

(a) `process_index` — Shorthand reference for future observer-to-NPCS communication; assigned and maintained by the NPCS.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.4.5. Errors

(a) No errors are returned for this call. The observer is blocked until this call successfully returns. This supports the start/restart policies described in section 13.8.

11.4.6. Engineering notes

(a) The process identifier (PID) must be returned in a operating system independent fashion.

(b) The `process_index` is used by the encapsulated library to determine which PMI/observer requested NPCS action using the PRI.

(c) Non-RPC implementations must be able to provide secure control and communication mechanisms if necessary. Not all IPC mechanism support a secure one-reader/*N*-writer model that is required for the NPCS and the *N* observers on the node.

(d) The lack of a properly executing NPCS must not reduce the availability or reliability of the instrumented DCE process.

(e) The instrumentation must not impact the instrumented process's execution state or functional behavior. The observer must invoke `dms_pri_register_process()` prior to invoking `dms_pri_register_sensor()`. This ensures proper behavior of the registration process in environments where all of DCE or DMS are not yet executing. In addition, an observer blocked in `dms_pri_register_process()`, or an observer that has not yet invoked `dms_pri_register_process()`, must not prevent sensors from calling their registration macros in a non-blocking fashion. The registration macros must enqueue the registration data so that it is available to the observer after it is un-blocked.

(f) An observer is the only element allowed to invoke the PRI routines. Sensors must use the sensor macros that will trigger out-of-line observer actions.

11.5. dms_pri_register_sensor()

11.5.1. Description

This function allows observers to provide the data to the NPCS to build the node level sensor registry. Standard and custom sensors within the process address space are registered by the observer using this function. The NPCS returns a sensor identifier that is used for all subsequent references to the registered sensor.

Sensors can be registered singly or in bulk. For efficiency, bulk registration should be used wherever possible. Since most DCE processes will contain dozens to hundreds of sensors, a bulk registration significantly reduces the RPC/IPC access overhead.

It is our assumption that the standard sensors (i.e., client, server, and global sensors) reside in DCE RTL, stubs, and DCE services (such as `secd` and `cdsd`). The custom sensors are those added by middleware components providers (such as Encina and DFS), and application client or server developers.

11.5.2. Function signature

```
error_status_t  dms_pri_register_sensor (
   [in    ] dms_process_index_t  process_index,
   [in,out] dms_instance_dir_t** sensor_register_list
  );
```

11.5.3. Function input

(a) `process_index` — Shorthand provided by NPCS via `dms_pri_register_process()`.

(b) `sensor_registration_list` — Specifies one or more sensors to register. Configuration data includes sensor name, sensor attributes and metric attributes.

11.5.4. Function output

(a) `sensor_registration_list` — The structure passed as input is returned with the sensor identifier and registration status fields set.

(b) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.5.5. Errors

Returned for entire call (i.e., summarizes results for all sensors that requested registration).

(a) `CHECK_INTERNAL_STATUS` — One or more sensors failed to register (see individual status for details). Check the status contained within the returned structure for details.

  (i) `registration_status` — Registration results for this particular sensor; one of:

    [a] `STATUS_OK` — Sensor registered with no problems.

    [b] `DUPLICATE_SENSOR` — Sensor already registered.

    [c] `ILLEGAL_NAME` — Sensor name not legal.

    [d] `ILLEGAL_CLASS` — Unknown sensor class.

    [e] `ILLEGAL_METRIC` — Unknown metric identifier.

(b) `UNKNOWN_PROCESS` — Process has never registered.

(c) `NO_NPCS` — NPCS not present. Unlike the `dms_pri_register_process()` function, the observer does not block if the NPCS is not present. On receipt of this error, the observer should initiate the restart policy described in section 13.8.

11.5.6. Engineering notes

(a) The observer in the instrumented DCE process should minimize the number of times it utilizes this expensive IPC mechanism by using bulk registration whereever possible.

(b) Standard sensor metric IDs must be defined and consistently maintained for each release of the nstrumentation system.

(c) Any PMA that requests a greater protection level then specified by the `minimum_protection_level` will have to decide whether to continue (see `dms_npmi_register_pma()`). The highest `minimum_protection_level` requested during the registration of sensors will be applied to ALL sensor data transported from this node to the PMA via the NPRI. This may cause excessive overhead, so use with caution.

(d) For an application that desires to support ''all or nothing'' semantics for registering a group of sensors, in the case of failure, all sensors with a `registration_status` of `STATUS_OK` should be immediately unregistered, using the `dms_pri_unregister_sensor()`.

(e) For non-RPC interfaces, the encapsulated library might generate a UUID and associate it with the sensor, and store it as its internal representation for the sensor identifier. Note that this specification does not require that the sensor identifier be unique for the cell, just unique for the node.

(f) Descriptive strings are necessary to name server interfaces and operations when presenting data to the end user. These `friendly names` require extensions to IDL to support a new structure in the stub or RTL that contains the string names. An API to retrieve these via the RTL must also be specified. The details of this are beyond the scope of the specification, but must be supported in the encapsulated library.

(g) Custom sensor registration requires a global repository for storing this data. The use of the DCE CDS to store metric name and instance, metric type, and help text is recommended. The utilities necessary to store this are beyond the scope of this specification. `metric_id` numbers for custom sensors must be unique within the process. This requires a utility function (not described in this spec), get_metric_id(), `that returns a unique metric_id each time it is invoked. Additional details regarding the need for a global repository are described in section 7.4.`

11.6. dms_pri_report_sensor_data()

11.6.1. Description

The observer uses this NPCS interface to report (push) modified sensor data during the last reporting interval. This allows the observer to report sensor data in an efficient manner, since it does not require the NPCS to poll for the next request and returns sensor data in bulk.

To speed up the performance of the steady-state path, it is not required that this function return errors synchronous with each call. Errors are guaranteed to be returned no later then by the next invocation of this function. Any data associated with bad status may be lost.

11.6.2. Function signature

```
error_status_t  dms_pri_report_sensor_data (
   [in    ] dms_process_index_t      process_index,
   [in    ] dms_observation_data_t*  sensor_report_list
  );
```

11.6.3. Function input

(a) `process_index` — Shorthand provided by NPCS via `dms_pri_register_process()`.

(b) `sensor_report_list` — One or more sensors and their component values are contained in this structure. See section 7.3.1 for additional details.

11.6.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.6.5. Errors

(a) `REPORT_FAILED` — Unknown error prevented NPCS from updating sensor data values (possible causes include lack of resources or execution time of NPCS).

(b) `NO_NPCS` — NPCS not present; observer should begin clean-up process.

11.6.6. Engineering notes

(a) The state diagram in Figure 10 shows the behavior of the observer with respect to providing data to NPCS. All the state transitions occur only at interval boundaries, with the exception of the NoMod → Config, and Data Modified → Config state transitions, which occur asynchronously with intervals. A copy of this state machine exists for each sensor. The input to the state machine is a modification flag set by probes and cleared by the observer. The objective is to report only non-zero or non-modified sensor data for an interval. This is in keeping with our philosophy to report only the minimum required data using the PRI and NPRI interfaces.

(b) To ease implementation of the encapsulated library and speed the performance of the steady-state path, it is not required for the function to return errors in a synchronous manner. It is only required that errors be returned at some future point in time (but no later then by the end of the next invocation of this function). Any data associated with bad status may be lost.

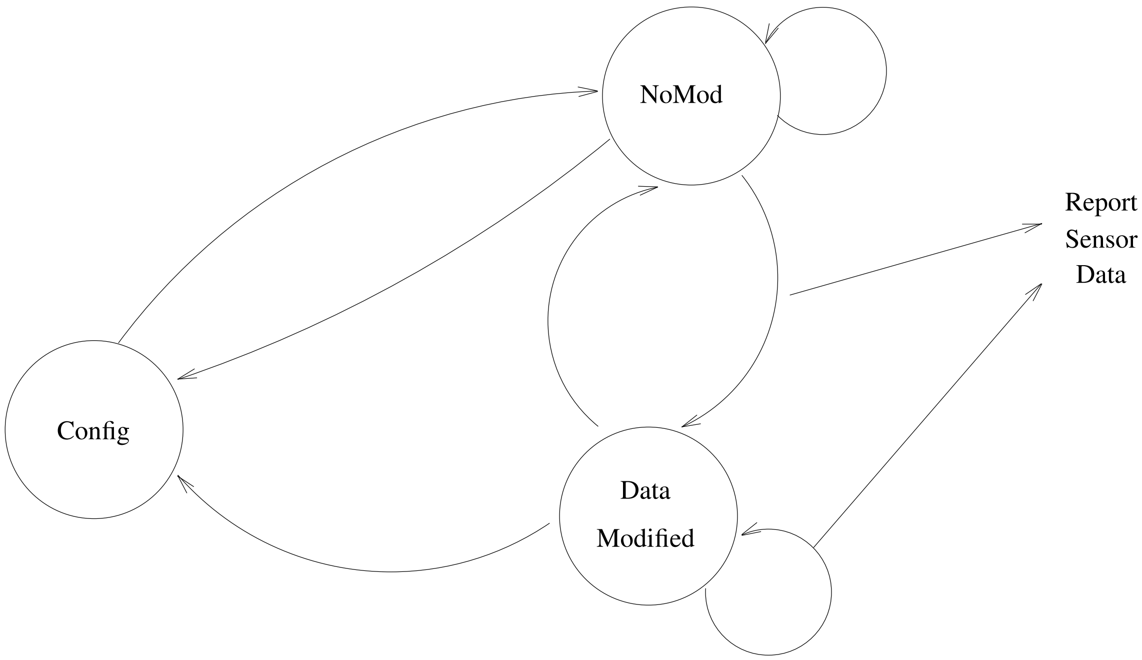

Figure 10. `dms_pri_report_sensor_data()` sensor state machine. Sensor data is pushed to the NPCS only if it was modified during the current reporting interval.

11.7. dms_pri_unregister_sensor()

11.7.1. Description

The observer uses this NPCS interface to notify the NPCS that one or more sensors can be removed from the node-level sensor registry. This allows the NPCS to free resources associated with these sensors.

In most cases, groups of sensors are unregistered only in the (unlikely) event of a server unregistering an interface.

11.7.2. Function signature

```
error_status_t  dms_pri_unregister_sensor (
   [in    ] dms_process_index_t  process_index,
   [in    ] dms_sensor_ids_t*    sensor_id_list
  );
```

11.7.3. Function input

(a) `process_index` — Shorthand provided by NPCS via `dms_pri_register_process`.

(b) `sensor_id_list` — A list of sensor identifiers to unregister.

11.7.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.7.5. Errors

(a) `NOT_REGISTERED` — One or more sensors were never registered.

(b) `NO_NPCS` — NPCS not present.

11.7.6. Engineering notes

(a) As sensors are unregistered the NPCS should use a ''recycling'' algorithm that does not attempt to re-use recently freed sensor identifiers. This will minimize the chance that PMAs will confuse cached but ''stale'' sensor identifiers with the incarnation of a new sensor.

11.8. dms_pri_unregister_process()

11.8.1. Description

An observer uses this NPCS interface to notify the NPCS to remove all of the sensors in the instrumented DCE process from the node-level sensor registry. This allows the NPCS to free resources associated with the unregistering process.

11.8.2. Function signature

```
error_status_t  dms_pri_unregister_process (
   [in    ] dms_process_index_t  process_index
  );
```

11.8.3. Function input

(a) `process_index` — Shorthand provided by NPCS via `dms_pri_register_process()`.

11.8.4. Function output

(a) *Function return value* — `dms_status` — Status of call; non-zero if call encountered an error.

11.8.5. Errors

(a) `NOT_REGISTERED` — Observer was never registered.

(b) `NO_NPCS` — NPCS not present.

11.8.6. Engineering notes

(a) None.

12. ADDITIONAL OBSERVER AND NPCS FUNCTIONS

This section describes additional functions supplied by the two standard mechanisms: the observer and the NPCS. Core functions were described in the relevant API sections. This section focuses on additional functionality necessary for the implementor of the measurement system to provide.

12.1. Observer Functions

Core observer functions were described in sections 7, 10 and 11.

The additional responsibilities are expressed in terms of an idealized implementation. It is possible that the responsibilities outlined here might require, or benefit from multiple observer threads.

(a) *Intervalized Capture of Raw Sensor Data.*

A snapshot of the raw data for each ''active'' sensor in an address space (process) must be made at the end of each summarization interval, by the data intervalizer executing on the observer thread. The notion of an `active` sensor is any sensor that has reached the end of its summarization interval, and has had execution of some thread pass through its final probe point during that interval (i.e., the sensor has produced some raw data from which its metric can be computed). This frees sensors from any direct responsibility for interval summarization, and provides the basis for time correlated metrics.

(b) *Computation of Intervalized Sensor Metrics.*

All sensor metric computations that are performed once per summarization interval are made on the snapshot raw data, by the metric calculator executing on the observer thread. This helps to minimize in-line sensor overhead. An example of this is the computation of mean response time, where the observer calculates the mean by dividing the cumulative response time by the number of completions.

(c) *Probes/Sensors for Process Global Sensors.*

Any interval sensor, i.e., a sensor that has probes executed once and only once each summarization interval, independent of any (''normal'') thread, will precede the data intervalizer's execution on the observer thread. This provide the means of supplying process global metrics that are independent of any other sensors, and minimizes overhead by collecting them out of the application's in-line path. Most of these sensors are described in section 5.6.

12.2. NPCS Functions

A Performance Management Application (PMA) is the ‘‘value-added’’ performance management and display application supplied by a vendor or third party. The PMA interacts with NPCS from across the network.

The NPCS is a ‘‘trusted process’’, but is used only for the collection and control of performance data. It should run as a non-privileged user.

Core NPCS functions were described in sections 7, 8, 9, 10 and 11.

(a) *Multiplexer.*

The NPCS is a many-to-one funnel for sensors on a node. It fulfils a similar function for the users of the data as well. While there may be many management stations wanting information, the NPCS buffers these requests so the sensors in the application server or client process do not have to manage multiple logical connections. The local sensor mechanism needs only to move the latest information to the (single) NPCS at the required rate, and for the requested information set. Then, NPCS will satisfy the various demands of the management stations requesting information. As such, it handles the state structures required to most efficiently assemble and move requested information to the performance management applications.

(b) *Unused State Recovery (garbage collection).*

NPCS may be implemented as a long-running daemon. Memory leaks in any form would be debilitating for a standard, required daemon. NPCS must have measures to identify sensors which have disappeared for whatever reason (e.g., process containing the sensors is killed or crashes). The memory and state associated with these sensors must be completely recovered. Similarly the state associated with defunct or disinterested PMAs must be recovered when the connection with the PMA is broken or unused.

(c) *LCD Time Management.*

As part of NPCS's role as multiplexer, it instructs the sensors in processes on the local node to report at the ‘‘least common denominator’’ (LCD) time interval to handle the requests from performance management applications. A bound would be selected that limits the time intervals that can be selected. For those performance management applications requesting relatively longer time intervals, NPCS summarizes multiple reports from the servers/clients reporting information on that node at the lower rate, and transmits only the data requested by the PMA. This is in keeping with our philosophy of transmitting the minimum data necessary across interfaces.

(d) *Transmitting Bulk Data for Efficiency.*

In the steady state, the NPCS will be supplying data to a PMA for several dozen, or even hundreds, of sensors. If each sensor is provided in a separate communication (RPC), the measurement system specification goals cannot be met. Thus the NPCS batches data at regular intervals from numerous sensors bound for a particular PMA.

(e) *Non-POSIX and Partial-DCE Implementations.*

On systems which are DCE-compliant, or which have some RPC mechanism of interest (but not truly DCE), a form of NPCS must be made available if data is to be collected. Perhaps in its translation capability, NPCS can be made available to management stations, even if running on a PC or non-POSIX operating system.

13. ENGINEERING ISSUES

This section documents all engineering issues related to the measurement system that were not described elsewhere in this document.

13.1. Conformance

The minimum functionality that is required to support this specification is:

(a) Standard sensors described in section 5 (custom sensor support is optional).

(b) NPMI API described in section 8.

(c) NPRI API described in section 9.

(d) PMI API described in section 10.

(e) PRI API described in section 11.

(f) The PMI/PRI encapsulated library mechanism described in section 7.1, or an RPC interface.

(g) Security as described in section 7.5.

(h) Internationalization as described in section 13.9.

(i) Supplemental observer functions described in section 12.1.

(j) Supplemental NPCS functions described in section 12.2.

13.2. Encapsulated Library

The requirements on the underlying implementation of the encapsulated library are that it correctly implements the various functions. A few points are emphasized here:

(a) ‘‘Who starts first’’ issues should be resolved so that the `dms_pri_register_process()` call by a process observer thread of an instrumented process will block until it has ‘‘executed’’ in the NPCS. The case of single-threaded DCE clients could be handled by immediately returning a ‘‘no NPCS yet’’ status. Checking the CMA value `cma__g_thrdcnt` can be used to determine multi-threaded support. Sensors being registered by other threads in the process will need to be queued for later registration with the NPCS, but these threads cannot be blocked because the NPCS may not ever appear.

(b) Since the library is emulating a procedure call mechanism, calls should be synchronous and return accurate status. The exception to this is `dms_pri_report_data()`. Because this is a bulk data transfer mechanism, it can return immediately, improving its efficiency. Note the caller of `dms_pri_report_sensor_data()` must be permitted to deallocate the input `dms_observation_data_t` data structures as soon as the call returns. This implies that either the return must be delayed, the data must be copied before returning, or some other (more complicated) PMI deallocation callback must be added if the underlying implementation permits, to allow more data to be queued. Errors may be reported later, on subsequent calls. Also, the possibility exists that a failing NPCS will cause a `dms_pmi_terminate()` callback, rather than bad status on a subsequent (*pri2*) call.

(c) When the NPCS or an instrumented process fails, the library should emulate a call (i.e., invoke the appropriate ‘‘server’’ procedure) to `dms_pmi_terminate()` or `dms_pri_unregister_process()`, to allow the other end to clean up.

(d) The observer thread must be prepared to replay its sensor registrations in the event of a crash, and restart of the NPCS. It should wait for the NPCS to restart by recalling `dms_pri_register_process()`.

(e) The library will need to monitor the process identifiers assigned by NPCS and returned by `dms_pri_register_sensor()`, in order to maintain a mapping from them to communications paths.

(f) The library will provide functions, `dms_pmi_el_free_outputs()` and `dms_pri_el_free_outputs()`, to handle the deallocation of output data structures. This permits the underlying memory management mechanisms to be the responsibility of the allocating module (NPCS, `npcs_lib`, `observer_lib`, DCE process). This also implies that in/out parameters need to be handled correctly to avoid memory leaks (i.e., save a copy of input pointers).

(g) The underlying IPC mechanism in this library must never block an entire process when used by either the NPCS or the observer. Blocking a single thread is acceptable.

(h) If an observer is blocked in `dms_pri_register_process()`, then sensors must be allowed to continue to invoke the registration macros. Individual sensor data is then enqueued until the observer is unblocked and able to process the sensor registration requests. The observer then processes these sensor registrations in bulk using the `dms_pri_register_sensor()` call.

13.3. DCE RTL

The DCE RTL needs to support a mechanism that allows client process to be identified and contacted if necessary for monitoring purposes.

Additional investigation is necessary to understand how to collect and report data for ‘‘nested RPCs’’ (i.e., an RPC that invokes a server, that causes the server to act as a client and invoke a different server).

13.4. DCE IDL

The DCE IDL must support a structure in the stub that contains data to construct ‘‘friendly named’’ sensors, since the RTL knows server operations by a UUID and an operation number (which is not very meaningful to a system administrator).

13.5. Other DCE Services

After the RTL is instrumented, all DCE core services should be recompiled to incorporate the instrumented `libdce`.

13.6. Sensor Information Sets

Since this capability is represented by a set, and individual sensors can support subsets, then it is a policy that all sensor data value components be returned in order of their definition in `dms_info_set_t`. If a particular sensor does not support a given set component, it must return NULL values in this sensor data value component location in `dms_sensor_data_t`. This also allows new set components to be defined and processed for future versions, as long as no set value is ever reused.

13.7. Application and DCE Availability

The non-existence or errors of the elements of the instrumentation must not decrease the availability of applications or DCE core services. This restriction reinforces the notion that the instrumentation is an aid to management, and not a hindrance.

13.8. Instrumentation Initialization and Restart

The instrumentation system must not decrease the availability of DCE applications or core services. Initialization and recovery of the measurement system are controlled to minimize impact on applications and core services. Thus this specification addresses a measurement system that supplements application and DCE core service functionality, and simplifies the design by eliminating recoverable data state mechanisms such as checkpoints.

Start-up dependencies are a crucial issue that must be addressed to ensure a robust implementation. An example of the problem illustrates the challenge: If the NPCS starts execution on a node prior to security or naming services, then the NPCS cannot provide secure communications (since this requires using a DCE login context that is not available without a security service). And if the NPCS on the same node as the security server starts execution after the security server, then the observer in the security process cannot register sensors (since this requires an NPCS supporting the PRI functions).

To resolve this dependency problem, a lazy connection strategy that allows elements to defer initialization and registration if the requested server component is not currently available is recommended. For the example in the previous paragraph, the security service defers registering sensors until the NPCS is available. The observer maintains registration context and periodically tests until the NPCS is available to complete registration. The NPCS has less of an issue since it responds to observer requests and does not initiate them. This technique has the benefit of allowing upgraded or failed NPCSs to be restarted in a live environment with no impact on application availability (although no performance data is available during the interval of NPCS inactivity).

Specifically, the following scenarios must be supported in conforming implementations. For each scenario the implementation policies are described.

13.8.1. Cell/node/process start-up

In this scenario, the cell, node, and instrumented DCE process start up for the first time.

Assumptions/requirements:

(a) NPCS requires access to security services and a DCE login context to support secure NPMI/NPRI functions.

(b) Security services registering sensors requires an executing NPCS.

Recommendation:

(a) Start the DCE core services in normal order.

(b) Observers within DCE core services block in `dms_pri_register_process()` since no NPCS exists on the node. While the observer is blocked, sensors must still be able to register within the process (but no calls to `dms_pri_register_sensor()` are allowed until the observer unblocks on `dms_pri_register_process()`). The observer is a separate thread, so there is no impact on the instrumented application.

(c) Start NPCS and authenticate with security service.

(d) Blocked observers are serviced by the NPCS, they unblock, and then they register sensors using the `dms_pri_register_sensor()` function.

(e) PMA can login, authenticate, and begin monitoring.

13.8.2. Node restart

In this scenario, the node is restarting after a planned or unplanned shutdown.

Assumptions/requirements:

(a) Sensors are initialized when processes restart.

(b) NPCS state of sensors on node is lost.

(c) PMA is not aware that node is restarted.

Recommendation:

(a) For NPCS, observer and sensor:

  (i) Follow policy in cell/node/process start up.

(b) For PMA:

  (i) PMA stops hearing from the NPCS via the NPRI. This does not apply to client-only PMAs (COPs), since they do not support the NPRI.

  (ii) Invoking any NPMI routine results in an RPC communication failure error (if the NPCS is not executing), or results in a ‘‘who-are-you’’ RPC status (if the NPCS has restarted but PMA not registered).

  (iii) PMA resets its internal sensor configuration state for all sensors on this node (since the observer will return all sensors to a quiescent state).

  (iv) After a user-configurable time, PMA re-registers with NPCS.

13.8.3. PMA terminate and restart

In this scenario, the PMA unexpectedly terminates and restarts. The NPCS and sensors are unaware of this event.

Assumptions/requirements:

(a) PMA state of sensors on node is lost.

(b) NPCS and sensors are not aware that PMA has failed/restarted.

Recommendation:

(a) NPCS invokes NPRI functions that result in an RPC communication failure. This does not apply to client-only PMAs, since they do not support the NPRI.

(b) After a user-configurable time:

(i) For non-COPs: The NPCS ceases to invoke NPRI routines and resets sensors configured only by this PMA to a quiescent state.

(ii) For COPs: Since there is no direct mechanism for the NPCS to test COP liveness, the NPCS periodically checks for when the last request was made by this PMA, and resets sensors configured only by this PMA to a quiescent state if the PMA has not made a recent request. The maximum period for client-only PMA inactivity is 7 days. This allows COPs to sample the instrumentation on a low-frequency basis, while minimizing resource consumption of the NPCS's internal tables.

(c) PMAs reregister after restarting.

13.8.4. NPCS shutdown and restart

In this scenario, the NPCS process gracefully exits.

Assumptions/requirements:

(a) NPCS's state of sensors on the node is discarded.

(b) PMA is not aware that NPCS has terminated.

(c) Observer and sensors are informed that NPCS has terminated.

Recommendation:

(a) For NPCS:

(i) NPCS invokes `dms_pmi_terminate()` prior to exiting. This informs the encapsulated library that the NPCS is no longer available.

(b) For observer and sensors:

(i) Same as NPCS shutdown/restart, described above.

(c) For PMA:

(i) Same as NPCS crash/restart, described below.

13.8.5. NPCS crash and restart

In this scenario, the NPCS unexpectedly terminates and restarts. The PMA and sensors are unaware of this event.

Assumptions/requirements:

(a) NPCS state of sensors on node is lost.

(b) PMA, observer and sensors are not aware that NPCS has failed/restarted.

Recommendation:

(a) For PMA:

(i) Same as node restart, described above.

(b) For observer:

(i) On the receipt of an PRI function that results in an error in communicating to the local NPCS, the encapsulated library must set a global flag that informs all observers that the NPCS has terminated.

Note that the encapsulated library must provide a synchronous mechanism to notify observers that the NPCS has terminated. Otherwise, an observer that is not currently reporting data will be ''lost'' and not reachable when the NPCS restarts.

(ii) Observers reset all sensors to a quiescent state.

(iii) Observers unregister (this must break the current connection with the encapsulated library, and clean up any encapsulated library state related to this observer).

(iv) Observers re-register. This is like the node start up.

(c) For NPCS:

(i) Same as node start up.

13.8.6. DCE process shutdown

In this scenario, the instrumented DCE process gracefully exits.

Assumptions/requirements:

(a) Sensors within the process are deleted.

(b) NPCS is informed of sensor deletions so that it can free resources.

(c) PMA is not informed of sensor deletions.

Recommendation:

(a) Observer invokes PRI unregister sensor function to communicate sensor termination.

(b) NPCS removes these sensors from its registry.

(c) PMA is informed implicitly by errors returned on explicit NPMI get and set operations.

(d) PMA removes these sensors from its registry.

13.8.7. DCE process crash and restart

In this scenario, the instrumented DCE process unexpectedly terminates.

Assumptions/requirements:

(a) Sensors within the process are deleted.

(b) NPCS is not informed of sensor deletions.

(c) PMA is not informed of sensor deletions.

Recommendation:

(a) Observer terminates before it can invoke PRI unregister sensor function. This requires that the encapsulated library provide an implementation dependent mechanism for detecting observers and sensors that are no longer executing.

(b) An encapsulated library dependent routine informs the NPCS of observer termination. The NPCS removes all observer’s sensors from its registry.

(c) PMA is informed implicitly by errors returned on explicit NPMI get and set operations.

(d) PMA removes these sensors from its registry.

(e) After instrumented DCE process is restarted, the situation is the same as cell/node/process start up.

13.8.8. Network partition

In this scenario, the PMA and NPCS are separated by a network partition.

Assumptions/requirements:

(a) Network partition is not directly detectable by neither PMA nor NPCS.

Recommendation:

(a) For the NPCS, same as the PMA crash/restart described above.

(b) For the PMA, same as the NPCS crash/restart described above.

13.9. Internationalization

Sensors contain cleartext descriptions that assist the end-user in interpreting the metric values. These descriptions are contained in a help text string. This string must support internationalization conventions as described in the various DCE RFCs on internationalization. Sensor names conform to the DCE portable character set.

13.10. Integration with Host OS; X/Open Data Capture Interface (DCI)

The DCI provides a standard interface to operating system performance data. The spec was submitted to X/Open in early 1994. That technology was evaluated for support by the functions in this specification. However, due to the concerns of availability and the uncertainty of the final shape of that standard, this specification does not explicitly support the DCI. But the following areas have been influenced by the DCI X/Open standard proposal:

(a) Namespace.

(b) Security.

(c) Node Data Communication and Storage (between observer and NPCS).

A list of DCE instrumentation requirements was provided to the authors of the DCI, for possible incorporation into the X/Open spec.

13.11. Instrumenting the Instrumentation System

It may be desirable to collect performance measures on the four APIs themselves. The activities associated with these APIs should not be included in the totals for the process. Optionally, they should be measurable by a PMA just like any other interface. The implementations of the observer, NPCS, and the four APIs must support self-instrumentation.

13.12. Design Rationale

This section describes several factors that influenced our design and recommendations.

13.12.1. Considerations of scale

The measurement infrastructure must perform efficiently over a wide range of network topologies and cell sizes. While our design supports monitoring across cells, the primary monitoring functions will align with the administrative domain of the cell. Table 2 illustrates the scale of the measurement system from a server perspective (clients are not included, although they represent a potentially larger pool). The table estimates the following quantities to gauge the demands placed on the measurement system (DCE specific terminology is used):

(a) The number of sensors per server manager operation.

(b) The number of manger operations per manager.

(c) The number of managers per server interface.

(d) The number of interfaces per server.

(e) The number of application servers per network node.

(f) The number of network nodes per DCE cell.

(g) E table then summarizes the two quantities.

(h) The number of sensors per network node.

(i) The number of sensors per DCE cell.

The number of operational sensors on a single node is large (500-8,000), and the number in a cell is very large (50,000-8,000,000 or more). (Note that transaction processing and distributed object applications may support a dozen or more interfaces. This may increase the actual number of sensors in a cell.) These estimates, however, are probably pessimistic with respect to the number of active sensors, since cells will contain a large number of different applications in different domains that are managed separately and therefore require fewer active sensors.

	''Typical'' Application	''Large'' Application
Sensors / Operation	10	20
Operations / Manager	5	10
Managers / Interface	1	1
Interfaces / Server	1	2
Server / Node	10	20
Nodes / Cell	100	1,000
Sensors / Node	500	8,000
Sensors / Cell	50,000	8,000,000

Table 2. Instrumentation Scale Considerations.

Having control over the sensor state is crucial for meeting measurement system overhead goals. This is accomplished by the end-user judiciously selecting the information sets for the sensors of interest. Only sensors of interest can be enabled and collected.

The above estimates do not include the number of active client sensors. This specification expects that only rarely will all clients have active instrumentation, due to excessive loading of node and network alike. To improve scalability of the measurement system it is expected that only a few clients are monitored at any time per application, in order to gather status and response times as proxies for others on the same node or in the same network. One final practical limitation for clients is that DCE does not support an identification mechanism for locating clients (only servers that register with the CDS).

13.12.2. Transporting data: pushing versus polling

A major implementation issue of the measurement system was whether to transport data by periodically ''pushing'' it across the network, or forcing PMA's to explicitly request or ''poll'' for data, similar to the SNMP philosophy. After significant discussion, it was decided to require NPCSs to push data to PMAs. Basically, the reasons why the push model was selected for implementation follows:

(a) *Scalability.*

Since the situation is really a large number of servers (sensors) pushing to a smaller number of NPCSs (e.g., 1 per system), which in turn pushes to a very small number of PMAs (maybe 1-10 per enterprise), then pushing scales better than polling potentially thousands of sensors to find only those with new data. In fact, keeping the amount of data sent small is very important for network utilization and scalability. Pushing also allows thresholds to be used, and significantly reduces the amount of data sent, even for the largest of systems.

(b) *State.*

In the push case, the pusher needs to keep state information about all its consumers (PMAs). It needs to know who, where and when. It also needs to know if a data item has not been delivered. Moreover, only the NPCSs know exactly when the data for the PMA is available. Storing this state is simpler for NPCSs, because of the small number of PMAs registered at any one moment.

In pull, the NPCSs would not be able to ignore the state information. Since no real saving in state is possible, the push case minimizes the state for PMAs. The PMAs will get cumulative data so they won't loose information if a sample is dropped, and they can tell if a sample is dropped or stale from timestamps.

(c) *Serialization.*

Although push is inherently serial, NPCSs can start multiple threads to push, but an NPCS thread is blocked during the push. (It may take some time for the PMA to respond.) Most important, since there are often practical limits to the number of active threads, the NPCSs would have very few active threads for push, while PMAs would have to have a large number of threads for parallel pulls. For scalability issues, NPCSs would have a limited pool of threads to push. There would normally be enough to dedicate one per PMA, but a pool would remove any hard limit.

(d) *Storage.*

This is an advantage since NPCS controls the flow of data, it can discard data that has been delivered to all interested parties. It also does not need to maintain a queue of requests. However, it does need to maintain a table of state information on ALL PMAs. In addition, the assumption was made that all data for a sample to a PMA would be packaged together into a single push.

(e) *Traffic.*

Because of the need to ensure (if not guarantee) delivery of the data to PMAs, the push is at least a data ACK pair. Pulls would require one more message. In addition, to minimize traffic, only data is sent and packaged into one response per sample to the PMA. A stateless pull (like NFS) would require state information in the pull, which increases traffic.

(f) *Scheduling.*

Since the sensors have an observer thread that is pushing to the NPCS, the timing of when to send the sample data to the PMA is only precisely known to the NPCS. That makes the scheduling of the data send time easy for the NPCS. Most important, for thresholds where data is only sent when a value is exceeded, the NPCS is the ONLY place that knows when this occurs, and that a data send is required. A pull would require the NPCS to wait and collect all the information anyway.

There is still an issue for scheduling of the PMA's data reduction, and correlation with the data arriving asynchronously from many NPCSs. However, since that is the highest level of the measurement system, and is the element with the least time sensitivity in the measurement system, it was considered an acceptable requirement. There may be several receiver threads, or one simply collecting data.

(g) *Error Handling.*

For the push model, data is flowing to the PMAs from the NPCSs. By providing timestamps and cumulative data, the PMAs can deal with missing data by either extrapolating, skipping, or another ‘‘make right’’ strategy. As far as dealing with failures, the NPCSs would know who and where they were sending data to, so the lack of a PMA ACK indicates a failed PMA, which allows the NPCS to free up resources belonging to that PMA.

NOTE: Even though the steady-state system is push-based, it was decided that a polling request function would be included in the NPMI to support special PMAs. This allows flexibility for something like a pull if used infrequently. The reason this is required is for SNMP support, client-only PMAs, and PMAs that register only thresholds but have not seen any data for awhile. A pull request allows the PMA to see the current data even if no thresholds were exceeded.

13.12.3. Sensor placement

This section describes sensor implementation issues and placement locations within the RPC runtime library (RTL).

The fundamental implementation question regards the placement of the sensors: Are they generated by the IDL compiler and placed in the stubs, or are they an integrated part of the DCE kernel (runtime library)?

Instrumenting the stubs using IDL has merit. Coupled with an internal tracing tool, these form very powerful application development/debugging utility. Unfortunately, for performance monitoring of arbitrary applications in a large environment, the IDL approach has several shortcomings.

First, sensors within stubs are visible to application developers, and thus modifiable by them. This is not safe for standard functions. Sensors within the RTL are not modifiable by the application writer. Second, supporting standard libraries is a pragmatic software engineering technique that minimizes implementation divergence in production environments. It also provides extensibility without the need to recompile an application's source code (users dislike recompilation because it almost always causes something to break). If sensors are in the RTL, then merely relinking the application with `libdce` provides new sensors. The requirement to relink (instead of recompile) also makes it easier to instrument other DCE services (CDS, Security) and middleware (Encina and CICS).

Other issues also influenced this direction. First is a lack of control over the granularity of collection (all or nothing), and the resulting deluge of data that is generated (especially for all clients) with a stub based architecture. (The scalability of this approach is unacceptably poor in a large environments.) The RTL is dynamically configurable to collect only the minimum amount of data that is requested. Finally, the need for pervasive support of this sensor requires a standard interface to sensors. Creating a standard performance interface to a stub is problematic.

Because of these arguments, we have chosen a hybrid implementation of the standard sensors. Most are located in the RTL but some are located in the stubs to capture stub specific processing.

13.12.4. Threshold detection

To minimize the amount of data transferred across the network counter and timer sensors, we support a threshold level detection mechanism. For example, a response time sensor set at the threshold would report data only when a user-configured threshold condition is TRUE (for example, when the maximum response time exceeds 20 seconds). In practice, we simplified the sensor implementation, and have the NPCS analyze the incoming data from the sensor to detect thresholds. This allows different PMAs to configure the same sensor with different threshold values, and still minimize the amount of data transported across the network. It is important to note that sensors report summarized data, thus the threshold detection is based on integrated values (mean, minimum or maximum) over a sampling interval.

13.12.5. Time units

Two distinct timer sensors, each with a different granularity, were proposed: seconds and nanoseconds. This will provide sufficient resolution, and future growth for the next 5-7 years. Note that overflow concerns may require that sum-of-squares terms have a coarser granularity.

To ensure timer resolution and efficient timestamp access, the spec defines a function that returns the time from the host OS with the proper granularity, and is implemented as efficiently as possible (this eliminates the problems with the POSIX `gettimeofday()` function). This implementation-specific routine is described in section 6.4.

13.12.6. Generic

IDL pickling is used to support pass-thru sensors. This results in several advantages:

(a) Allows for large unknown data structures.

(b) Allows for adding sensors with arbitrary data without requiring a modification to this specification.

(c) Allows the observer to transmit data without knowing about the sensor's structure.

The use of pickling results in several issues:

(a) *Efficiency* — Don't slow down sensor reporting by adding overhead for pickling if the pickling is not necessary. The specification has provided a keyed union to allow for generic sensor data: long value, array of long values, and opaque bytes (may be used for pickling).

(b) *Registering Metrics* — It has been proposed that the pickling information be sent across with the data. But, in order for the pickled data to be any use to the PMA, the PMA must have been compiled with the header file (probably output by an IDL compilation of sensor pickling functions). Thus the PMA must already have an idea of which custom metrics it plans to use.

13.13. Future Items for Specification

The following items have been deferred for a future working group:

(a) Investigate how to provide sensors for PC clients running Windows95 and NT. This may require supporting an interface to the Desktop Management Interface.

(b) Histograms providing distribution frequencies for a monitored event. They are not supported in this version of the specification, but are a candidate for future support as a natural extension of the sensor information set.

(c) Event tracing can provide explicit cause-and-effect for application characterization. Sensors to support this capability need to be investigated.

(d) Resource accounting and charge-back are crucial management functions. This document describes a specification for measurement that can be extended to support resource accounting. We strongly recommend that a future resource accounting system NOT be designed with redundant measurement infrastructure, since this will only result in increased overhead.

(e) There is a need to optimize the notification of reporting changes to the NPCS sensor registry. We debated between two alternatives: a mechanism that would create a version number for each unique version of the registry, and allow queries using a comparison version number; or a mechanism that would notify PMAs of modifications that impact the current configured sensors.

(f) Multiple views of sensors: Although the instrumentation name space is organized in a hierarchy, there are may circumstances in which a consumer of instrumentation data will want to group the data in different ways. A system administrator might, for example, want to simultaneously observe the performance of all machines on which security daemons are running, or the CDS daemons which serve a particular clearinghouse. This specification does not provide an explicit mechanism for doing this; we believe that the definition and maintenance is a function best left to the individual performance management applications which will make use of the data this specification describes. At the same time, we hope that developers of performance management applications will develop common mechanisms for storing and transferring group definitions, so that users of different applications will be able to observe the same data with a minimum of manual re-configuration.

(g) Extend `dms_pri_register_sensor()` to allow a process to specify its minimum data protection level, to automatically control the RPC data protection level used for PMA and NPCS communication. This feature eases system administration by allowing application clients or servers to establish the protection level during the development phase.

(h) All DCE core services should be instrumented (the CDS metrics are described in RFC 32.0 [RFC 32]), to capture logical events and other service specific concerns.

(i) The performance measurement interface should become part of a standard server management interface that is available for all DCE-based processes.

(j) The authors' collective experience with previous projects has led them to conclude that software instrumentation is subject to the second law of thermodynamics: Over time, the instrumentation tends towards a more disordered state. This disorder is a result of defect repair and new functionality that changes the behavior of the instrumented software, and consequently the precise location (and meaning) of the instrumentation probe points. This has significant ramifications on maintaining the accuracy and the utility of instrumentation. To resolve this a validation suite to certify the instrumentation must be defined and implemented.

A validation suite is required to ensure the correctness of the initial implementation of the sensors, and to provide a test case to demonstrate future correctness. Furthermore, an interoperability test for the interfaces is required to ensure interface compatibility.

14. ACKNOWLEDGMENTS

This document is the result of many individuals who contributed their time and expertise.

Rich Friedrich, Joe Martinka, Steve Saunders, Gary Zaidenweber, Tracy Sienknecht, Dave Glover (Hewlett-Packard Company).

Dave Bachmann, Ellen Stokes, Robert Berry (International Business Machines, Inc.).

Barry Wolman, Dimitris Varotsis, David Van Ryzin (Transarc).

Sarr Blumson (CITI (Center for Information Technology Integration), University of Michigan).

Art Gaylord (Project Pilgrim, University of Massachusetts).

15. REFERENCES

[CMG] Computer Measurement Group — Performance Management Working Group, *Requirements for a Performance Measurement Data Pool*, Revision 2.3, May 1993.

[Laz] E. Lazowska, et al, *Quantitative System Performance*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1984.

[RFC 11] M. Hubbard, *DCE SIG Serviceability Requirements Document*, OSF DCE-RFC 11.0, August 1992.

[RFC 32] R. Friedrich, *Requirements for Performance Instrumentation of DCE RPC and CDS Services*, OSF DCE-RFC 32.0, June 1993.

[RFC 38] *DME/DCE Managed Objects Requirements Document*, OSF DCE-RFC 38.1, 1994 (to appear).

[Rose] M. Rose, *The Simple Book — An Introduction to Management of TCP/IP Based Internets*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.

APPENDIX A. dms_binding.idl

```
[
 version(2.2)
]
interface dms_binding
/*
 * This interface defines the data structures used to represent
 * relationships between entities (sensors/processes/nodes) within
 * DMS.  Some are "transparent", meaning that a user of that
 * structure can manipulate its contents.  Some are "opaque", meaning
 * that only the creating entity can manipulate its contents.
 */
{
/* TRANSPARENT BINDING TYPES */

typedef [string] unsigned char dms_string_t[];

typedef unsigned long  dms_protect_level_t;  /*see rpc.h*/

typedef [string] unsigned char  dms_string_binding_t[];

/* OPAQUE BINDING TYPES */

typedef unsigned long  dms_pma_index_t;

typedef unsigned long  dms_npcs_index_t;

typedef unsigned long  dms_process_index_t;

typedef unsigned long  dms_sensor_id_t;

typedef struct dms_sensor_ids {
                     unsigned long    count;
    [size_is(count)] dms_sensor_id_t  ids[];
  } dms_sensor_ids_t;
}
```

APPENDIX B. dms_config.idl

```
[
 version(2.3), pointer_default(ptr)
]
interface dms_config
/*
 * This interface defines the sensor configuration data structures
 * for specifying the configuration of individual sensors.
 */
{
import "dms_binding.idl", "dms_data.idl", "dms_status.idl";

const  unsigned long  dms_NO_METRIC_COLLECTION      = 0;
const  unsigned long  dms_THRESHOLD_CHECKING        = 0x00000001;
const  unsigned long  dms_COLLECT_MIN_MAX           = 0x00000002;
const  unsigned long  dms_COLLECT_TOTAL             = 0x00000004;
const  unsigned long  dms_COLLECT_COUNT             = 0x00000008;
const  unsigned long  dms_COLLECT_SUM_SQUARES       = 0x00000010;
const  unsigned long  dms_COLLECT_SUM_CUBES         = 0x00000020;
const  unsigned long  dms_COLLECT_SUM_X_TO_4TH      = 0x00000040;
const  unsigned long  dms_CUSTOM_INFO_SET           = 0x80000000;

typedef unsigned long dms_info_set_t;

typedef struct dms_threshold_values {
    dms_datum_t lower_value;
    dms_datum_t upper_value;
  } dms_threshold_values_t;

typedef union dms_threshold
    switch (boolean have_values) {
    case TRUE:
        dms_threshold_values_t values;
    case FALSE:
        ;
  } dms_threshold_t;

typedef struct dms_config {
    dms_sensor_id_t  sensor_id;
    dms_timevalue_t  reporting_interval;  /*0 == infinite*/
    dms_info_set_t   info_set;
    dms_threshold_t* threshold;
    error_status_t     status;
  } dms_config_t;

typedef struct dms_configs {
    unsigned long                   count;
    [size_is(count)] dms_config_t   config[];
  } dms_configs_t;
}
```

APPENDIX C. dms_data.idl

```
[
 version(2.2), pointer_default(ptr)
```

```
]
interface dms_data
/*
 * This interface defines the data structures that represent the
 * (sensor & attribute) data values communicated through DMS.
 */
{
import "dms_binding.idl", "dms_status.idl";

typedef struct dms_opaque {
    unsigned long        size;
    [size_is(size)] byte bytes[];
  } dms_opaque_t;

typedef enum {
    dms_LONG,
    dms_HYPER,
    dms_FLOAT,
    dms_DOUBLE,
    dms_BOOLEAN,
    dms_CHAR,
    dms_STRING,
    dms_BYTE,
    dms_OPAQUE,
    dms_DATA_STATUS
  } dms_datum_type_t;

typedef union dms_datum
    switch (dms_datum_type_t type) {
    case dms_LONG:
        long long_v;
    case dms_HYPER:
        hyper hyper_v;
    case dms_FLOAT:
        float float_v;
    case dms_DOUBLE:
        double double_v;
    case dms_BOOLEAN:
        boolean boolean_v;
    case dms_CHAR:
        char char_v;
    case dms_STRING:
        dms_string_t *string_p;
    case dms_BYTE:
        byte byte_v;
    case dms_OPAQUE:
        dms_opaque_t *opaque_p;
    case dms_DATA_STATUS:
        error_status_t status_v;
  } dms_datum_t;

typedef struct dms_sensor_data {
    dms_sensor_id_t                sensor_id;
    unsigned long                  count;
    [size_is(count)] dms_datum_t   sensor_data[];
```

```
  } dms_sensor_data_t;

typedef struct dms_timevalue {
    unsigned long sec;
    unsigned long usec;
  } dms_timevalue_t;

typedef struct dms_observation_data {
    dms_timevalue_t                      end_timestamp;
    unsigned long                        count;
    [size_is(count)] dms_sensor_data_t* sensor[];
  } dms_observation_data_t;

typedef struct dms_observations_data {
    unsigned long                             count;
    [size_is(count)] dms_observation_data_t* observation[];
  } dms_observations_data_t;
}
```

APPENDIX D. dms_naming.idl

```
[
 uuid(5e542624-e9d6-11cd-a3a9-080009273eb9),
 version(2.2), pointer_default(ptr)
]
interface dms_naming
/*
 * This interface defines the data structures that represent the dms
 * namespace.  There are two forms of names that can be represented,
 * a simple string only form and a fully decorated form.
 */
{
import "dms_binding.idl", "dms_data.idl", "dms_status.idl";

typedef struct dms_name_node*  dms_name_node_p_t;

typedef struct dms_name_nodes {
    unsigned long                      count;
    [size_is(count)] dms_name_node_p_t names[];
  } dms_name_nodes_t;

typedef struct dms_name_node {
    dms_string_t*     name;  /*"*" == wildcard*/
    dms_name_nodes_t  children;
  } dms_name_node_t;

typedef struct dms_attr {
    dms_string_t* attr_name;
    dms_datum_t   attr_value;
  } dms_attr_t;

typedef struct dms_attrs {
    unsigned long    count;
    [size_is(count)] dms_attr_t* attrs[];
  } dms_attrs_t;
```

```
typedef struct dms_sensor {
    dms_sensor_id_t  sensor_id;
    dms_attrs_t*     attributes;
    unsigned short   count;
    [size_is(count)] small metric_id[];
  } dms_sensor_t;

typedef struct dms_instance_leaf {
    unsigned long                    count;
    [size_is(count)]  dms_sensor_t* sensors[];
  } dms_instance_leaf_t;

typedef struct dms_instance_node*  dms_instance_node_p_t;

typedef struct dms_instance_dir {
    unsigned long                           count;
    [size_is(count)] dms_instance_node_p_t children[];
  } dms_instance_dir_t;

typedef enum {
    dms_DIRECTORY,  dms_LEAF,  dms_NAME_STATUS
  } dms_select_t;

typedef union dms_instance_data
    switch (dms_select_t data_type) {
   case dms_DIRECTORY:
     dms_instance_dir_t*  directory;
   case dms_LEAF:
     dms_instance_leaf_t* leaf;
   case dms_NAME_STATUS:
     error_status_t         status;
  } dms_instance_data_t;

typedef struct dms_instance_node {
    dms_string_t*        name;
    dms_datum_t*         alternate_name;
    dms_instance_data_t  data;
  } dms_instance_node_t;
}
```

APPENDIX E. dms_npmi.idl

```
[
 uuid(e8f6e46e-e9d7-11cd-be13-080009273eb9),
 version(2.2), pointer_default(ptr)
]
interface dms_npmi
/*
 * This interface defines the operations provided to a PMA by a NPCS.
 * The interface can by utilized by two styles of PMA, full-function
 * and client-only PMA.  A full function PMA must support the
 * dms_npri interface, and can either have sensor data pushed to it,
 * or pull sensor data from a NPCS.  The client-only PMA (COP) will
 * not support the dms_npri interface, and must pull sensor data from
 * a NPCS.
```

```
 */
{
import "dms_status.idl", "dms_binding.idl", "dms_data.idl",
       "dms_config.idl", "dms_naming.idl";

error_status_t  dms_npmi_register_pma (
   [in    ] handle_t               handle,
   [in,ptr] dms_string_binding_t*  npri_binding,
                                   /*null == client-only PMA*/
   [in    ] dms_npcs_index_t       npcs_index,
   [in    ] dms_protect_level_t    requested_protect,
   [   out] dms_pma_index_t*       pma_index,
   [   out] dms_protect_level_t*   granted_protect
  );

[idempotent]
error_status_t  dms_npmi_get_registry (
   [in    ]  handle_t              handle,
   [in    ]  dms_pma_index_t       pma_index,
   [in,ptr]  dms_name_nodes_t*     request_list,
                                   /*null == entire registry*/
   [in    ]  long                  depth_limit,
                                   /*0 == infinity*/
   [   out]  dms_instance_dir_t**  registry_list
  );

error_status_t  dms_npmi_set_sensor_config (
   [in    ]  handle_t         handle,
   [in    ]  dms_pma_index_t  pma_index,
   [in,out]  dms_configs_t**  sensor_configs
  );

error_status_t  dms_npmi_get_sensor_data (
   [in    ]  handle_t                   handle,
   [in    ]  dms_pma_index_t            pma_index,
   [in    ]  dms_sensor_ids_t*          sensor_id_list,
   [in    ]  boolean                    bypass_cache,
   [   out]  dms_observations_data_t** sensor_data
  );

error_status_t  dms_npmi_unregister_pma (
   [in    ]  handle_t            handle,
   [in    ]  dms_pma_index_t     pma_index
  );
}
```

APPENDIX F. dms_npri.idl

```
[
 uuid(ee7599b2-e9d7-11cd-8e49-080009273eb9),
 version(2.2), pointer_default(ptr)
]
interface dms_npri
/*
 * This interface defines the operation provided to a NPCS by a PMA
```

```
 * to received sensor data from that NPCS.  This interface is not
 * provided by a client- only PMA (COP).
 */
{
import "dms_status.idl", "dms_binding.idl", "dms_data.idl";

[idempotent]
error_status_t  dms_npri_report_sensor_data (
    [in    ] handle_t                  handle,
    [in    ] dms_npcs_index_t          npcs_index,
    [in,ptr] dms_observations_data_t* sensor_data
                                       /*null == keep-alive*/
  );
}
```

APPENDIX G. dms_pmi.idl

```
[
 local, version(2.2)
]
interface dms_pmi
/*
 * This interface defines the operations provided to a NPCS by the
 * encapsulating library (npcs_lib).  Additionally the operations
 * that must be provided to npcs_lib by a NPCS are specified.
 */
{
import "dms_status.idl", "dms_binding.idl", "dms_data.idl",
       "dms_config.idl", "dms_naming.idl";

typedef [ref] error_status_t (*dms_pri_reg_proc_fp_t) (
                        [in] dms_string_t*        process_name,
                        [in] long                 process_pid,
                       [out] dms_process_index_t* process_index
                                  );

typedef [ref] error_status_t (*dms_pri_reg_sensor_fp_t) (
                    [in] dms_process_index_t  process_index,
                    [in] dms_protect_level_t  min_protect_level,
                [in,out] dms_instance_dir_t** sensor_register_list
                              );

typedef [ref] error_status_t (*dms_pri_report_data_fp_t) (
                    [in] dms_process_index_t      process_index,
                    [in] dms_observation_data_t*  sensor_report_list
                        );

typedef [ref] error_status_t (*dms_pri_unreg_sensor_fp_t) (
                    [in] dms_process_index_t  process_index,
                    [in] dms_sensor_ids_t*    sensor_id_list
                        );

typedef [ref] error_status_t (*dms_pri_unreg_proc_fp_t) (
                    [in] dms_process_index_t  process_index
                        );
```

```
/*
 * The following functions are needed to encapsulated the dms_pmi and
 * dms_pri interfaces in a library (npcs_lib).
 */

error_status_t  dms_pmi_el_initialize (
   [in    ] dms_pri_reg_proc_fp_t     pri_register_process,
   [in    ] dms_pri_reg_sensor_fp_t   pri_register_sensor,
   [in    ] dms_pri_report_data_fp_t  pri_report_sensor_data,
   [in    ] dms_pri_unreg_sensor_fp_t pri_unregister_sensor,
   [in    ] dms_pri_unreg_proc_fp_t   pri_unregister_process
  );

error_status_t  dms_pmi_el_free_outputs (
   [in,ptr] dms_configs_t*          sensor_config_list,
                                    /*null == absent*/
   [in,ptr] dms_observation_data_t* sensor_report_list
                                    /*null == absent*/
  );

/*
 * The following functions provide the basic dms_pmi functionality.
 */

error_status_t  dms_pmi_set_sensor_config (
   [in    ] dms_process_index_t  process_index,
   [in,out] dms_configs_t**      sensor_config_list
  );

error_status_t  dms_pmi_get_sensor_data (
   [in    ] dms_process_index_t      process_index,
   [in    ] dms_sensor_ids_t*        sensor_id_list,
   [   out] dms_observation_data_t** sensor_report_list
  );

error_status_t  dms_pmi_terminate (
   void
  );
}
```

APPENDIX H. dms_pri.idl

```
[
 local, version(2.3)
]
interface dms_pri
/*
 * This interface defines the operations provided to an instrumented
 * process by the encapsulating library (observer_lib).  Additionally
 * the operations that must be provided to observer_lib by an
 * instrumented process are specified.
 */
{
import "dms_status.idl", "dms_binding.idl", "dms_data.idl",
       "dms_config.idl", "dms_naming.idl";
```

```
typedef [ref] error_status_t (*dms_pmi_set_config_fp_t) (
                     [in] dms_process_index_t process_index,
                 [in,out] dms_configs_t**     sensor_configs
                         );

typedef [ref] error_status_t (*dms_pmi_get_data_fp_t) (
                     [in] dms_process_index_t      process_index,
                     [in] dms_sensor_ids_t*        sensor_id_list,
                    [out] dms_observation_data_t** sensor_report_list
                         );


typedef [ref] error_status_t (*dms_pmi_terminate_fp_t) (
                          void
                              );
/*
 * The following functions are needed to encapsulated the dms_pri and
 * dms_pmi interfaces in a library (observer_lib).
 */

error_status_t  dms_pri_el_initialize (
   [in    ] dms_pmi_set_config_fp_t  pmi_set_sensor_config,
   [in    ] dms_pmi_get_data_fp_t    pmi_get_sensor_data,
   [in    ] dms_pmi_terminate_fp_t   pmi_terminate
  );

error_status_t  dms_pri_el_free_outputs (
   [in,ptr] dms_instance_dir_t* sensor_register_list
                                /*null == absent*/
  );

/*
 * The following functions provide the basic dms_pri functionality.
 */

error_status_t  dms_pri_register_process (
   [in    ] dms_string_t*        process_name,
   [in    ] long                 process_pid,
   [   out] dms_process_index_t* process_index
  );

error_status_t  dms_pri_register_sensor (
   [in    ] dms_process_index_t  process_index,
   [in,out] dms_instance_dir_t** sensor_register_list
  );

error_status_t  dms_pri_report_sensor_data (
   [in    ] dms_process_index_t      process_index,
   [in    ] dms_observation_data_t*  sensor_report_list
  );
   /*Note: return (status) may correspond to previous call!*/

error_status_t  dms_pri_unregister_sensor (
   [in    ] dms_process_index_t  process_index,
   [in    ] dms_sensor_ids_t*    sensor_id_list
```

```
  );

error_status_t  dms_pri_unregister_process (
   [in    ] dms_process_index_t  process_index
  );
}
```

APPENDIX I. dms_status.idl

```
[
 version(2.4)
]
interface dms_status
/*
 * This interface defines the set of (resulting) status values for
 * all the operations and data structures defined in DMS.
 */
{
import "dce/nbase.idl";

const error_status_t    dms_STATUS_BASE       = 0x114b2001;
const error_status_t    dms_STATUS_OK         = error_status_ok;
const error_status_t    dms_NOT_IMPLEMENTED   = dms_STATUS_BASE + 0;
const error_status_t    dms_UNKNOWN_SENSOR    = dms_STATUS_BASE + 1;
const error_status_t    dms_UNKNOWN_PROCESS   = dms_STATUS_BASE + 2;
const error_status_t    dms_UNKNOWN_INFO_SET  = dms_STATUS_BASE + 3;
const error_status_t    dms_UNKNOWN_THRESHOLD_LEVEL
                                              = dms_STATUS_BASE + 4;
const error_status_t    dms_UNKNOWN_NPCS      = dms_STATUS_BASE + 5;
const error_status_t    dms_UNKNOWN_PMA       = dms_STATUS_BASE + 6;
const error_status_t    dms_ILLEGAL_NAME      = dms_STATUS_BASE + 7;
const error_status_t    dms_ILLEGAL_METRIC    = dms_STATUS_BASE + 8;
const error_status_t    dms_ILLEGAL_SENSORID  = dms_STATUS_BASE + 9;
const error_status_t    dms_ILLEGAL_VALUE     = dms_STATUS_BASE + 10;
const error_status_t    dms_ILLEGAL_BINDING   = dms_STATUS_BASE + 11;
const error_status_t    dms_SENSOR_CONFIG_CONFLICT
                                              = dms_STATUS_BASE + 12;
const error_status_t    dms_SENSOR_NOT_CONFIGURED
                                              = dms_STATUS_BASE + 13;
const error_status_t    dms_SENSOR_NOT_MODIFIED
                                              = dms_STATUS_BASE + 14;
const error_status_t    dms_DUPLICATE_SENSOR  = dms_STATUS_BASE + 15;
const error_status_t    dms_NO_SENSOR_REQUESTED
                                              = dms_STATUS_BASE + 16;
const error_status_t    dms_NO_NPCS           = dms_STATUS_BASE + 17;
const error_status_t    dms_NO_THRESHOLD      = dms_STATUS_BASE + 18;
const error_status_t    dms_REPORT_FAILED     = dms_STATUS_BASE + 19;
const error_status_t    dms_FUNCTION_FAILED   = dms_STATUS_BASE + 20;
const error_status_t    dms_NOT_REGISTERED    = dms_STATUS_BASE + 21;
const error_status_t    dms_REGISTER_FAILED   = dms_STATUS_BASE + 22;
const error_status_t    dms_ALREADY_REGISTERED
                                              = dms_STATUS_BASE + 23;
const error_status_t    dms_PROTECT_LEVEL_NOT_SUPPORTED
                                              = dms_STATUS_BASE + 24;
const error_status_t    dms_BYPASS_NOT_ALLOWED
```

```
                                           = dms_STATUS_BASE + 25;
const error_status_t    dms_NO_OUTPUTS_FREED  = dms_STATUS_BASE + 26;
const error_status_t    dms_CHECK_INTERNAL_STATUS
                                           = dms_STATUS_BASE + 27;
const error_status_t    dms_BAD_STATUS        = dms_STATUS_BASE + 28;
}
```

AUTHORS' ADDRESSES

Rich Friedrich
Hewlett-Packard Company
1501 Page Mill Road, Mailstop 1U-14
Palo Alto, CA 94304
USA
Internet email: richf@hpl.hp.com
Telephone: +1-415-857-1501

Steve Saunders
Hewlett-Packard Company
11000 Wolfe Road, Mailstop 42U
Cupertino, CA 95014
USA
Internet email: saunders@cup.hp.com
Telephone: +1-408-725-8900

Gary Zaidenweber
Hewlett-Packard Company
300 Apollo Drive
Chelmsford, MA 01824
USA
Internet email: gaz@ch.hp.com
Telephone: +1-508-256-6600

Dave Bachmann
International Business Machines, Inc.
11500 Burnet Road, MS 9132
Austin, TX 78758
USA
Internet email: bachmann@austin.ibm.com
Telephone: +1-512-838-3170

Sarr Blumson
CITI, University of Michigan
519 W William
Ann Arbor, MI 48103
USA
Internet email: sarr@citi.umich.edu
Telephone: +1-313-764-0253