A Proposal for DASADA
Gauge Infrastructure Working Group

DRAFT V 1.0

June 2, 2001

ABLE Research Group
Carnegie Mellon University

1. Introduction

The DARPA Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA) Program is predicated on the principle that we can make systems far more robust by adding capabilities of self-observation and self-adaptation. Specifically, a system that can observe its own behavior or other properties, and then take corrective actions to improve itself, will be far less brittle than current statically-configured systems.

An essential part of this vision is the notion of *probes* and *gauges.* Informally, probes are responsible for collecting low-level observations of a system, while gauges are responsible for abstracting and interpreting information from probes (or other gauges). Thus, by analogy with monitoring and control of physical systems, probes play the role of sensors in a physical system, while gauges play the role of dials, warning lights, and visual monitors. Probes are deployed in some physical system and provide data in terms of that physical universe (e.g., software units, networks, processors). Gauges provide information about some abstraction, or *model*, of the system (e.g., an architectural model).

The distinction between “low-level” (probe) and “high-level” (gauge) observations is, of course, a bit arbitrary. In a degenerate case one could imagine a very smart probe that both monitors a system and provides an interpretation of it. But in general we expect that the two technologies will be implemented by different parties: run time monitoring experts will develop probe technologies that require detailed knowledge of a system’s implementation and operating platform. In contrast, gauge developers will build technologies that interpret observed data in terms of various abstract models (such as architectural models, behavior models, security models, etc.) and their properties.

Given the breadth of possible values that might be gauged, we expect that there will be a wide variety of gauges. Their active phases vary from design-time to deployment-time and run-time. Some gauges may be computation-intensive, while others mainly aggregate and translate lower-level information; some gauges are interested in a system’s code structure, while others may report the execution status of a running system or information about its run-time environment.

In order to achieve a form of plug and play among all these diverse probes and gauges, it is necessary to have standards that will facilitate interoperability. A proposal for a probe run-time infrastructure has been proposed: it defines what it means to be a probe in terms of the kind of interface it must support and the expected run time behavior that it will exhibit.

This proposal attempts to do something similar for gauges. Specifically, it defines (1) a common framework for describing, developing and integrating gauges, which can be used as a standard that is shared between gauge developers and gauge consumers/integrators; and (2) a common set of services that support run-time communication between gauges and the consumers of their outputs. Our goals are (a) to simplify gauge development and integration by providing generic (and complete) gauge specification requirements and an API, and (b) to support cross-platform communication between gauges and consumers.

This document consists of five sections. We first introduce the overall design of the gauge infrastructure and provide a motivating example. In Section 2, we define gauges and their functions. Section 3 provides a method for specifying gauges more formally. In Section 4, we define the gauge run-time infrastructure, and the APIs for gauges and their consumers. Various open issues are enumerated in Section 5.

1.1 Background

Detailed, dynamically-gathered measurement information is essential to ensure the composability, dependability and adaptability of software systems. Without such information, it will not be possible to determine whether a system should be changed, or to locate points of failure that need to be fixed.

There are three essential parts of the story. First is *monitoring*: there must be some way to observe the status of a system (whether a design- or run-time). Second is *interpretation*: monitoring information must be interpreted in the context of some system features or properties of interest. Third is *reconfiguration*: when problems are detected the system must be adjusted.

In this report we focus on the second of these parts: the gauge infrastructure is proposed as part of the measurement solution that automatically and dynamically collects, aggregates, analyzes and reports system information.

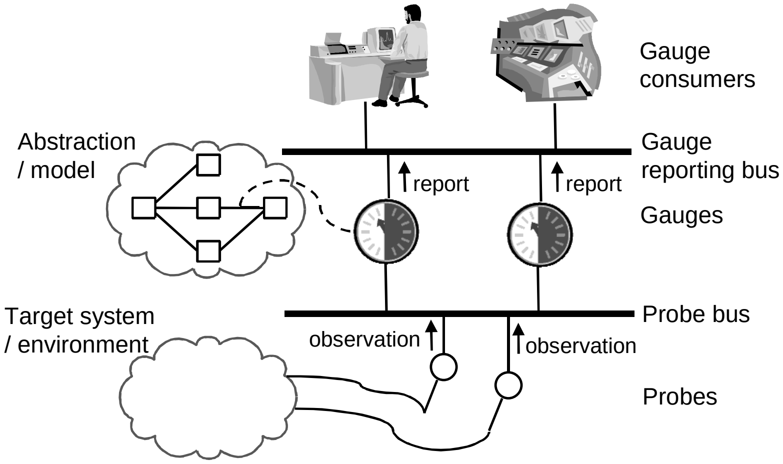

Figure 1. The gauge and probe infrastructure

The overall conceptual architecture is illustrated in Figure 1. At the lowest level is a set of *probes*, which are "deployed" in the target system or physical environment, and announce observations of the actual system via a "probe reporting bus." At the second level is a set of *gauges*, which consume and interpret lower-level probe measurements in terms of higher-level model properties. Like probes, gauges disseminate information via a “gauge reporting bus.” The top-level entities in Figure 1 are *gauge consumers*, which consume information disseminated by gauges. Such information can be used, for example, to update an abstraction/model, to make system repair decisions, to display warnings and alerts to system users, or to show the current status of the running system.

1.2 Example

Consider the simple client/server file transfer system *Simple_CS* illustrated in Figure 2, consisting of a client *C*, a server *S*, and a connector *L* connecting *C* and *S*. A latency gauge *G* is associated with the system’s architectural model to dynamically report *L*’s latency property. (In an Acme description the interpreted value is represented as *L*’s latency property, *L.Latency*.) In order to measure the latency, *G* needs to know the IP addresses of the hosts running *C* and *S*, which can be either determined statically at design time, or bound to the IP addresses of *C* and *S* at run time. The IP addresses required by *G* are “setup parameters” to be provided upon gauge creation.

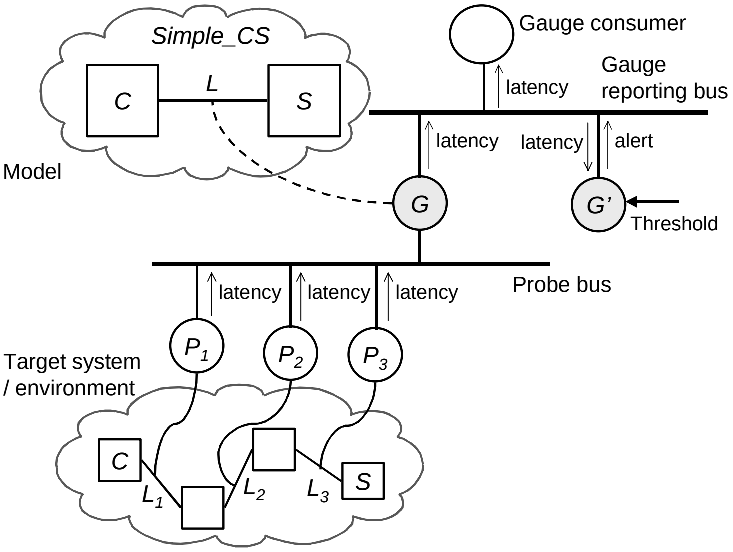

Figure 2. Gauge examples

Let us assume that in the implementation of this system, *L* consists of multiple physical links. *G* needs to deploy some probes (P_1 through P_3 in this case) to these physical links in order to measure the latency of each physical link, gather the latency values reported by these probes, and derive the composite latency of *L*. After deriving that latency, *G* announces the value over the gauge reporting bus. Consumers of *G*’s events can then obtain *L*’s latency value at run time, and take various actions, such as displaying the value, evaluating whether the observed latency is problematic, or deciding whether the connector *L* needs to be upgraded. In this example, we will use another gauge, *G’*, that compares the latency value reported by *G* with a threshold value that can be set at creation time, or through its control interface. *G’* will then generate an alert whenever *L.Latency* exceeds this threshold value – an event that can in turn be consumed by others.

2. Gauge Definition

Gauges are software entities that gather, aggregate, compute, analyze, disseminate and/or visualize measurement information about software systems. Software tools/agents, software engineers and system operators consume such information, use it to evaluate system state and dynamically make adaptation decisions. In its pure form, a gauge does not change its associated model or control the software system directly. However, the outputs of a gauge may be used by other entities to effect such changes.

Several principles or assumptions underlie our notion of gauges and have been used to guide the design of gauge specification and gauge APIs. These assumptions include:

1. *A gauge should be able to gather information from probes and/or other gauges.* In most cases, gauges will deploy probes and generate information based on probe values that is meaningful in the context of some model. However, in some cases it may make sense for a gauge to generate values that are not based on probes or other gauges (for example, timer gauges).
2. *The value (or values) reported by a gauge can have multiple consumers*. A single gauge consumer can use multiple gauges. For example, there may be gauge consumers that simply monitor and report values to the user, and other consumers that automatically detect impending failure and take actions to adapt the underlying system automatically, but use the same model as the basis for both activities. In this case, we do not want to duplicate gauges.
3. *A gauge should not know the identity or number of its consumers.* Gauge consumers can collaborate, but they are not required to know of each other.
4. *Different types of gauges will be developed by different parties.* We expect there to be a wide variety of gauge types, reflecting the diverse needs for system monitoring and adaptation. We expect that in many cases a heterogeneous mix of gauges will be operating in a distributed fashion on multiple (heterogeneous) platforms.
5. *The set of gauge consumers can change dynamically.* In this way we can dynamically adapt our monitoring infrastructure to add new observational capability as needed.
6. *A gauge can be controlled at run time to set various parameters of its operation.* Examples include parameters to control the frequency of reporting, or the parts of the system that is being monitored.
7. *Each gauge has a type, which describes the gauge's setup and configuration requirements, and the types of values that it reports.* Gauge developers and gauge consumers should have a contract that specifies what to provide and require from a gauge.
8. *Gauges are associated with models.* Models allow gauges to interpret their inputs and produce higher-level outputs. Moreover, gauge values must be meaningful in some context, and the model provides the context. For example, the latency gauge of the example above, interprets the physical observations in terms of an abstract connector in the context of a specific architectural model.
9. *A gauge instance is associated with at most one model.* However, instances of the same type gauge can be associated with different types of models For example,

the information produced by a latency gauge might be associated with an architectural model (e.g., in Acme or C2) or a UML model, but a specific gauge instance cannot be associated with two different models.

10. *Values reported by a gauge should be self-describing.* By this we mean that a reported gauge value should contain enough information for a consumer to interpret the value without needing to obtain more information (the model used to interpret the value, for example).

We also identify the need for certain gauge administrative entities – called *gauge managers* – that will be developed to facilitate the control, management, and meta-information query of gauges. These entities are also part of the gauge run time infrastructure, and are described in more detail below.

2.1 Gauges vs. Probes

As mentioned above, the distinction between gauges and probes is somewhat arbitrary. Therefore it is helpful to list the key *qualitative* differences between gauges and probes, so that based on the different usages and requirements, developers can choose to implement their monitoring software as a gauge or as a probe.

The primary difference is that gauges are associated with a system model, whereas probes are deployed in an actual system. A probe developer/user needs to consider when and how to deploy and install it, and thus requires detailed system implementation and operation knowledge. On the other hand, a gauge developer/user is typically interested in some system model. Typically a gauge will hide the details of probe deployment and operation from the consumer of the more abstract information provided by the gauge.

A gauge is associated with a particular model to help assign meaning to the values it produces, and how those values should be interpreted. For example, a gauge producing a value for latency may only be producing that value in the context of a particular Acme model, and would likely not have the same meaning in a Meta-H model of the same system. Models may be as complex as architectural descriptions or as simple as a set of threshold values that can be used to determine whether the system is in a safe, dangerous or critically-impaired state. The model provides the context in which the value is interpreted. This association between gauges and models is a key distinguishing feature from probes.

The information reported by a gauge is mapped to properties in a model, while the information reported by a probe corresponds to properties of the target system. For example, in Figure 2, probe P_1 reports the latency of the physical link L_1, while G reports the latency of the abstract connector L. The value reported by P_1 is invisible in the architectural model and need not be understood in this model. In general, gauges interpret low-level observations in the context of the high-level models.

2.2 Functions of Gauges and Gauge Managers

Based on the example in Figure 2, a gauge must support the following functions:

(1) A gauge's primary function is to report values corresponding to the properties of a system model. In the previous example, the latency gauge *G* reports a value associated with connector *L*'s latency property in *Simple_CS*' architectural model.

(2) A gauge can be configured when it is created or afterwards. For example, the latency gauge *G* can be configured to collect and report information at different frequencies. The smart gauge *G'* can be configured with different threshold values.

(3) A gauge can be queried about its current state. Such information is useful for consumers that have been created after the corresponding gauges have been created or configured. For example, the state of *G* could include the IP addresses it was connected to, and the sampling rate at which it is producing values.

(4) A gauge needs to be created before being used, and an be removed at some time later. The gauge manager provides lifecycle and other gauge management services, as yet undefined.

(5) A gauge manager can provide meta-information such as names and types of values that a particular gauge type reports, as well as information about the gauge's configuration parameters. Such introspection support provides extra flexibility for gauge integration, and is supported by the gauge manager in this proposal.

Optionally, a gauge can collect information from other gauges and probes. These functions are listed below:

(6) A gauge can register its interest in values reported by probes, control these probes, and query their meta-information;

(7) A gauge can register its interest in values reported by other gauges, control these gauges, and query their states or meta-information.

We group these functions into orthogonal capabilities supported by different entities:

- The gauge interface, which includes gauge reporting, gauge control, and gauge query. A gauge can optionally be a probe consumer and/or a gauge consumer.
- The gauge manager interface, which supports meta-information query and gauge lifecycle management capabilities.

In Section 4 we address the gauge, gauge consumer and gauge manager interfaces in detail. The probe consumer interface is described elsewhere in the "Probe Run-time Infrastructure" documentation.

3. Gauge Specification

Given the diversity of gauges, implemented by many different parties, using different programming languages, running on different hardware and software platforms, it is important to be able to characterize gauges so that a system builder can determine what types of gauges are available and what kinds of capabilities that type of gauge has. Such a characterization could also be used by gauge developers as a functional specification around which to base their implementations, and by the gauge run-time infrastructure to manage gauges by providing gauge meta-information. In this section we consider how one might specify a gauge. In brief, a gauge's specification describes (1) its associated model (and model type), (2) the types of values that it reports and the associated model properties, and (3) setup and configuration parameters.

To keep the gauge specification as generic as possible, this document only defines the requirements of a gauge specification, (i.e., what a gauge specification should describe), not a particular syntax for that specification. We expect that various model developers will provide specific syntax for specifying gauges for those models.

3.1 Type versus Instance Specifications

Each gauge has a type. A gauge *type specification* describes the shared features of instances of a gauge type. A gauge *instance specification* defines a particular gauge. A gauge instance includes information about the gauge that elaborates the gauge type specification and associates the outputs of the gauge with a particular abstract model or elements of a model. For example an instance of the latency gauge type, illustrated in Figure 2, would identify the IP address set-up parameters, a default "frequency of sampling" control parameter, and indicate the model and connector for which it is calculating a latency value.

3.2 Gauge Type Specifications

A gauge type specification is a tuple consisting of the following parts:

1. The name of the gauge type: for example, *Latency_Gauge_T*;
2. The set of values reported by the gauge (specified using a name and a type): for example, the *Latency_Gauge_T* reports one value, *Latency* of type *float*;
3. Setup parameters (including name, type, and default value for each parameter): for example, the *Latency_Gauge_T* has two setup parameters: *Src_IP_Addr* and *Dst_IP_Addr*, which are both of type *String* and have no default value;
4. Configuration parameters (including name, type, and default value for each parameter): for example, the *Latency_Gauge_T* has one configuration parameter *Sampling_Frequency*, which is of type *milliseconds* with a default value of *50.* The sets of configuration parameter and setup parameter are not necessarily disjoint. A default value should be provided for each configuration parameter that is not in the set of setup parameters.[1]
5. Comments: these explain in more detail what a gauge does and how to interpret the values (the values' units, accuracies, etc). which provide more detail on the functionality of the gauge.

Table 1 describes a gauge type for the gauge G in Figure 2, that measures latency value in milliseconds, represented as a floating point number.

Gauge Type	`Latency_Gauge_T`
Reported Values	`Latency: float`

Setup Parameters	`Src_IP_Addr: String [default=""]`
	`Dst_IP_Addr: String [default=""]`

Configuration Parameters	`Sampling_Frequency: int [default=50]`
Comments	`Latency_Gauge_T measures network latency of a connector whose endpoints are defined by a source and destination IP address.`

Table 1: An example of gauge type specification

---

[1] Currently only literal values are allowed for setup and configuration parameters.

3.3 Gauge Instance Specifications

A gauge instance specification is a tuple consisting of the following parts:

1. The name and type of the gauge instance: for example, *G* is the name of the latency gauge in Figure 2, which is of gauge type *Latency_Gauge_T*;
2. The name and type of the model that the gauge is associate with: for example, *G* is associated with a model called *Simple_CS*, which is of type *Acme*;
3. Mappings from values reported by the gauge to the associated model properties. Each mapping is a tuple of <*GaugeValue*, *ModelProperty*>, meaning that the *GaugeValue* actually reflects the value of *ModelProperty*;
4. Setup values: these can be statically specified or dynamically provided upon gauge creation. If no value is provided, the default value of this gauge type should be used;
5. Configuration values: these can be statically specified or provided at run-time. If no value is provided, the default value of this gauge type should be used.
6. Comments: to describe more details of the gauge's function.

Table 2 specifies the gauge instance *G* that we discussed in the previous example.

Gauge Name: Gauge Type	`G: Latency_Gauge_T`
Model Name: Model Type	`Simple_CS: Acme`
Mapping	`<Latency, L.Latency>`

Setup Values	`Src_IP_Addr = L.IP1;`
	`Dst_IP_Addr = L.IP2;`

Configuration Values	`Sample_Frequency = 100`
Comments	`G is associated with the L Connector of the system, Simple_CS, defined as an Acme model.`

Table 2: An example of gauge instance specification

In most cases, the tuple of <*GaugeType, GaugeName, ModelType, ModelName*> can be used as a unique gauge instance identifier. Naming issues are discussed later in more detail.

3.4 The Role of a Gauge Specification

A gauge type specification can be written by a gauge consumer developer and submitted to interested gauge developers. The gauge developers may implement gauges and gauge managers that satisfy the specification, and deploy the gauge managers. Meanwhile, the gauge consumer developers will model their systems and associate gauge instances with the model (gauge instance specification). Based on the system model, they can implement the system and write code to create the associated gauges and interact with them.

On the other hand, a gauge developer can write a gauge type specification and submit it to a gauge repository. Gauge consumer developers can search for gauges in the repository and use them in their systems.

The gauge type specification can be used by gauge developers as a gauge functionality specification. It also provides meta-information to be managed by the gauge manager. The gauge instance specifications provide information for gauge consumer developers to create gauges and translate gauge reporting.

A guage consumer can use the gauge type name to locate a gauge manager. The gauge name, gauge type, model name, and model type can useful in registering interest in values reported by a gauge

4. Gauge Run-time Infrastructure

4.1 The Gauge Infrastructure Architectural Style

The gauge infrastructure consists of a gauge reporting bus and different kinds of gauges, gauge managers, and gauge consumers. The gauge reporting bus is the primary communication medium between gauges and gauge consumers. Gauges publish reports to the gauge reporting bus. Gauge consumers subscribe to such information and are notified by the gauge reporting bus.

Gauge managers provide gauge lifecycle management to create and remove gauges, and meta-information query services for gauge introspection. The gauge reporting bus and gauge managers are available at run time, and constitute the gauge run time infrastructure.

Figure 3 illustrates the gauge infrastructure style by indicating types of components and their interfaces in the gauge infrastructure architecture, as well as the kinds of connectors over which they can interact. (Appendix A provides an Acme description of this architectural style.) There are three types of components in the architecture: gauge, gauge manager, and gauge consumer. Gauges and gauge managers provide services through their interfaces (as discussed later).

Defining a standard set of interfaces for these entities in the gauge infrastructure will provide the interoperability and plug-compatibility required when using diverse implementations technologies to monitor systems. For each component, its interfaces include both what the component is required to provide, and what services can be used by the component. For example, a gauge manager is required to provide a gauge creation service, and can use the event publishing/subscription service provided by the gauge reporting bus. A gauge consumer is required to subscribe its interest in some gauge reporting, and can create, delete, or configure gauges.

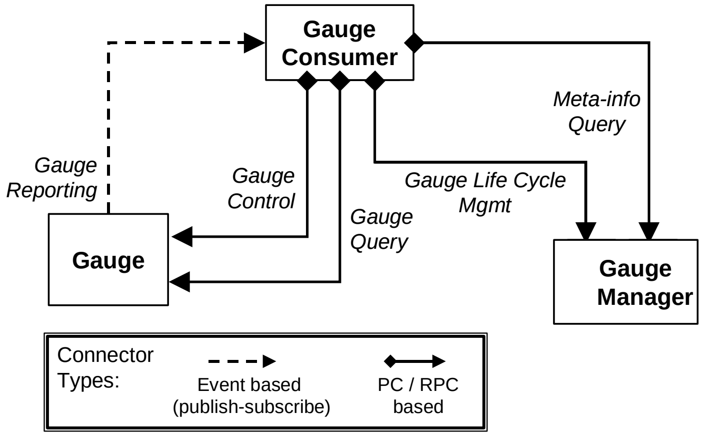

Figure 3. The logical architecture of Gauge infrastructure: Components and their interfaces

Logically,[2] two different communication models (connector types) are used in the architecture: event-based (or publish/subscribe), and remote procedure call (RPC). The gauge reporting interface uses event-based communication, publishing events without knowing which interested parties will receive them. The other gauge interfaces are all related to control and query functions, whose users must have knowledge of the target entity (i.e., a specific gauge or gauge manager), and require a reply from the target. RPCs or procedure calls are used by these interfaces.

To test the feasibility of this design, and provide a starting point for gauge developers, we have implemented a prototype of the gauge run-time infrastructure that we will refer to as the Gauge Infrastucture "reference implementation". The implementation is in Java, and a gauge reporting bus built on top of Siena Event Bus, some example gauges, and their corresponding gauge managers. Details about the reference imlementation are covered in Section 4.3.

4.2 Component/Connector Interfaces

We now discuss the interfaces to the gauge reporting bus, gauges, gauge managers, and gauge consumers. A component interface is described as a group of functions with brief explanations. Each parameter is associated with one of three modifiers to flag whether it provides input via the API (`in`), returns a value (`out`), or is a combination of both (`inout`).

For RPC-based functions, we use handles to hide the details of how to maintain the peer-to-peer interaction between a local caller (or client) and a remote callee (or server). A handle is used as a local unique reference to a possibly remote object. Thus a handle is a proxy object that acts as if it is the actual object it refers to. For example, the gauge creation function called by a gauge consumer returns a handle to a gauge object. The gauge consumer can call the gauge handle's procedures to perform operations on the corresponding gauge.

The handle takes care of any remote interaction with the actual gauge, which may be located remotely. Upon creation, a handle stores the identifier of the remote object, and maintains the state of interaction when communicating with the remote object. The actual communication between the handle and its corresponding object can be based on existing RPC mechanism, an event bus, or some other communication mechanism.

To deal with the failure of a gauge or gauge manager interface call some error reporting mechanism is required. However, such mechanisms are typically language specific. In this document we simply specify a boolean value of "*false*" or a *null* reference to indicate an occurrence of error. Specific implementations might use more detailed error return codes or exceptions.

In some situations, multiple values must be passed as input or output parameters. We will use two types of vectors as data containers for this purpose. A StringPairVector is an array in which each element is a pair of strings, like <attribute, value>. A

---

[2] At this architectural design level for the gauge infrastructure it is important to pick connector abstractions that closely match the intended modes of communication. However, nothing prohibits the underlying implementation from using other lower-level communication mechanisms to implement these connectors. Indeed, as we will see, the reference implementation uses the Siena publish-subscribe bus as the transport mechanism for *both* event-based and RPC-based communication.

StringTripleVector is an array in which each element is a tuple of triple strings, like <attribute, type, value>.

4.2.1 The Gauge Reporting Bus

A gauge reporting bus provides a gauge-infrastructure-specific, value-reporting channel between gauges and gauge consumers. The interface of a gauge reporting bus includes the following functions:

1. Subscribe for events that match the given constraints.

   ```
   boolean subscribeInterest(in String eventType,
                 in String gaugeName, in String gaugeType,
                 in String modelName, in String ModelType,
                 in GaugeConsumer interestedParty)
   ```

   The *interestedParty* is registered for all events that match the constraints provided by the parameters. For the parameters, *eventType* is a constraint used to filter events, currently it must be one of the following values: “reportValue”, “reportMultipleValue”, “reportCreated”, “reportDeleted”, “reportConfigured”. Similarly, *gaugeName*, *gaugeType*, *modelName*, *modelType*, are also event filtering constraints, and only events that match the constraints will be delivered to the *interestedParty*. A string value of “*” is treated as a wildcard (i.e., no constraint). *interestedParty* is a *GaugeConsumer* object, which in turn needs to provide corresponding event handling functions that will be called when new notifications come in (see below). The return value indicates whether the subscription was successful.

2. Unsubscribe to events.

   ```
   boolean unsubscribeInterest(in String eventType,
                 in String GaugeName, in String gaugeType,
                 in String modelName, in String ModelType,
                 in GaugeConsumer interestParty)
   ```

   The *interestedParty* un-subscribes for all events that match the constraints. The event bus will not deliver such events to the *InterestedParty* in the future. The return value indicates whether the unsubscription was successful.

When a gauge consumer subscribes to a certain type of events, a corresponding filter for such events is created and associated with the consumer. The filter is uniquely determined by the combination of *eventType*, *gaugeName*, *gaugeType*, *modelName*, and *modelType* parameters. A wildcard can be used for any of these constraints, except that we do not all five parts to be wildcards, which will subscribe for nothing. When the gauge consumer unsubscribes its interest in certain events, any previous subscription that matches the unsubscription.[3]

3. Report a single value

   ```
   boolean reportValue(in String UID,
   ```

---

[3] In the current reference implementation an unsubscription must exactly match a subscription in order for the removal to succeed. This is a consequence of using the Siena implementation as the publish-subscribe substrate.

```
        in String gaugeType, in String gaugeName,
        in String modelType, in String modelName,
        in String valueName, in String propertyName, in String value)
```

The values of *gaugeType*, *gaugeName*, *modelType*, *modelName* provide descriptive information that allows interested parties to filter events of interest. An observation about *valueName*, which corresponds to model property *propertyName* with a value of *value*, is made by the announcer and will be translated into a report on the gauge reporting bus. A return value of *false* indicates that the event publishing action failed.

4. Report multiple values

```
boolean reportMultipleValues(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in String modelName,
        in StringTripleVector values)
```

The values of *gaugeType*, *gaugeName*, *modelType*, *modelName* provide descriptive information that allows interested parties to filter events of interest. The array *values* contains tuples of <*valueName*, *propertyName*, *value*>. Each tuple represent an observation about *valueName*, which corresponds to model property *propertyName*, with a value of *value*. The array is translated into an event notification and published on the gauge reporting bus. A return value of *false* indicates that the event publishing action failed.

5. Report creation status

```
boolean reportCreated(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in String modelName)
```

Reports the creation of a gauge identified by the tuple <*UID*, *gaugeType*, *gaugeName*, *modelType*, *modelName*>. A return value of *false* indicates that the event publishing action failed.

6. Report deletion status

```
boolean reportDeleted(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in String modelName)
```

Reports the deletion of a gauge identified by the tuple <*UID*, *gaugeType*, *gaugeName*, *modelType*, *modelName*>. A return value of *false* indicates that the event publishing action failed.

7. Report configuration status

```
boolean reportConfigured(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in String modelName,
        in StringPairVector configParams)
```

Reports the configuration status of a gauge. This occurs typically after a gauge is reconfigured. The parameter *configParams* is an array of <*configParamName*, *configParamValue*> pairs. Each pair provides a current configuration value for the gauge *g*. A return value of *false* indicates that the event publishing action failed.

4.2.2 The Gauge Lifecycle

Figure 4 illustrates the lifecycle of a gauge. When a gauge manager is requested to create a new gauge, the gauge manager will allocate resources and use the provided setup parameters to create the gauge. Upon successful creation, the gauge will announce its creation, and transition into a “created & active” state. It is now ready to be configured or queried, and to report values. The gauge can be reconfigured in this state. After any reconfiguration it will announce the effect of configuration. Finally, a gauge can be deleted by a gauge manager. The effect of successfully deleting a gauge is that the gauge will announce its deletion, the gauge will generate no further reporting, and future control and query requests to the gauge will fail.

Compared with the lifecycle of probes (as described in the “Probe Run-time Infrastructure” documentation), the gauge lifecycle is a bit simpler. Currently it is not clear whether we should include more states/stages in a gauge lifecycle.

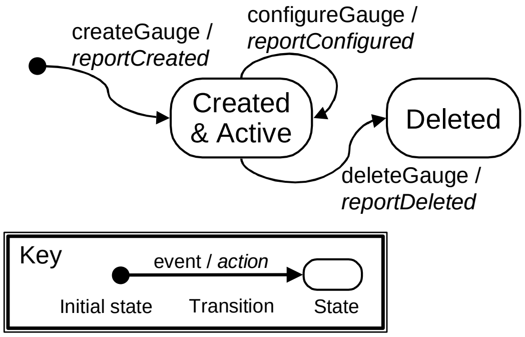

Figure 4. Gauge state machine

4.2.3 The Gauge Interface

As noted earlier, a gauge handle provides a gauge interface that encapsulates details of remote invocation. When a gauge is created by a gauge manager a gauge handle is returned to the gauge creator. Other interested gauge consumers can notice the existence of the gauge by observing announcements of gauge creation/configuration/deletion. They can then get a handle to the gauge from an appropriate gauge manager by using its unique identifier.

The gauge interface includes the following functions:

1. Create a gauge handle

   ```
   GaugeHandle(in String gaugeUID)
   ```

   A new gauge handle is returned. The handle refers to a gauge uniquely identified by *gaugeUID.*

2. Configure a gauge

   ```
   boolean configureGauge(in StringPairVector configParams)
   ```

   When this function is applied to a gauge handle its corresponding gauge will be configured using the given parameters. The parameter *configParams* is an array of <*configParamName*, *configParamValue*> pairs. Each pair provides a configuration value for the gauge. A return value of *false* indicates an occurrence of error.

3. Query the current gauge state

```
boolean queryGaugeState(out StringPairVector setupParams,
                        out StringPairVector configParams,
                        out StringPairVector mappings)
```

The gauge corresponding to the gauge handle will reply with its current state (including its setup parameters, current configuration, and properties corresponding to its report values). The parameters are three arrays of pairs, which will be filled in with pairs of the form <*setupParamName*, *setupParamValue*>, <*configParamName*, *configParamValue*> and <*valueName*, *propertyName*>, respectively. A return value of *false* indicates an occurrence of error.

4. Query the recent value reported by the gauge

```
boolean querySingleValue(in String valueName, out String value)
```

The gauge corresponding to the gauge handle will reply with its most recent value with a name of *valueName*. A return value of *false* indicates an occurrence of error.

5. Query the recent values reported by the gauge

```
boolean queryAllValues(out StringTripleVector values)
```

The gauge corresponding to the gauge handle will reply with all of its most recent values. The parameter *values* will be filled in with tuples of <*valueName, propertyname, value*>. A return value of *false* indicates an occurrence of error.

Finally, a gauge must report events on the gauge reporting bus via the GaugeReportingBus reporting interface.

4.2.4 Gauge Manager

The primary motivation for including gauge managers in the gauge infrastructure architecture is to provide a gauge lifecycle service, similar to a CORBA object factory. Specifically, a gauge manager knows how to create new gauge instances and allocate resources for them.

Given the diversity of possible gauges, we expect that there will be a wide variety of gauge managers. Some of them may be generic, providing management facilities for several types of gauges. Others may be gauge-type-specific (e.g., to be provided by the gauge developer), and manage a specific type of gauge. A gauge manager might be implemented as a central global server, or as multiple distributed collaborating servers. We leave the choice of implementation approaches to their developers.

A gauge manager provides meta-facilities such as gauge creation, gauge destruction, and gauge meta-information query. Other possible functions of gauge managers might be gauge name/UID management, gauge registry, and gauge factory discovery. This proposal defines a minimal set of services that a gauge manager must provide.

To gauge consumers, the gauge manager interface is provided by gauge manager handle, which include the following functions:

1. Get the handle of a gauge manager that provides services related to certain gauge type.

   ```
   GaugeMgrHandle connect(in String gaugeType)
   ```

   A gauge manager handle will be returned. The handle refers to a gauge manager that provides meta services related to gauges of type *gaugeType* A return value of *null* indicates that no gauge manager of *gaugeType* could be located.[4]

2. Create a new gauge using the given parameters.

   ```
   GaugeHandle createGauge(in String gaugeName,
          in String modelName, in String modelType,
          in StringPairVector setupParams, in StringPairVector mappings)
   ```

   A gauge instance will be created and a handle to the gauge will be returned. The actual gauge is associated with a model of *modelName* of type *modelType*. The type of the gauge instance is the same as the *gaugeType* parameter provided when getting the gauge manager handle. The name of the gauge is *gaugeName*, which can be used in combination with *gaugeType*, *modelType*, and *modelName* as event filtering constraints. The setup parameter array *setupParamters* is an array of <*setupParamName*, *setupParamValue*> pairs. Each pair provides a setup value for the created gauge. *Mappings* is an array of <*valueName*, *propertyName*>. Each pair represents a mapping from *valueName* to *propertyName.* After creation, gauge consumers can then subscribe to events generated by the gauge. A return value of *null* indicates an occurrence of error.

3. Delete a gauge

   ```
   boolean deleteGauge(in GaugeHandle gauge)
   ```

   The gauge corresponding to the gauge handle will be destructed. A return value of *false* indicates that the deletion action failed (the state of the gauge is unclear).

4. Create a new gauge using given parameters

   ```
   boolean queryGaugeMetaInfo(
                out StringPairVector configParamsMeta,
                out StringPairVector valuesMeta)
   ```

   The gauge manager referred to by the handle is queried about gauge type-related meta-information. The *valuesMeta* is a vector of pairs of the form <*valueName*, *valueType*>. Each pair represents a value that can be reported by this gauge type. The *configParamsMeta* is a vector of pairs of the form <*configParamName*, *configParamType*>. Each pair represents a possible configuration parameter. The corresponding gauge manager will handle this query, and response with *valuesMeta* and *configParamsMeta*. A return value of *false* indicates an occurrence of error.

---

[4] In the reference implementation appropriate gauge managers are located by publishing a gauge manager discovery event on the Siena Bus.

4.2.5 Gauge Consumers

A gauge consumer is not required to provide any functions, but it is free to call the functions provided by gauges, gauge managers, and the gauge reporting bus.[5] However, to be a gauge consumer, it must be subscribe to gauge reporting events and process the notifications from the gauge reporting bus.

Different consumers might have different event filters for gauge reporting. For example, most will be interested in values reported by a specific gauge, some will be interested in all values related to a model, and some will be only interested in values reported by gauges of particular type. Such interests can be expressed by using the subscription interface of the Gauge Reporting Bus.

In order for a consumer to "listen" to a the Gauge Reporting Bus, it must provide a set of callbacks to be called when events of interest are detected by the Bus. The implementation of these callbacks will vary from consumer to consumer.

1. Handle a report about a single value reported by a gauge

```
void onReportValue(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in string modelName,
        in String valueName, in String propertyName, in String value)
```

2. Handle a report about multiple values reported by a gauge

```
void onReportMultipleValues(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in string modelName,
        in StringTripleVector values)
```

3. Handle a report about gauge creation

```
void onReportCreated(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in string modelName)
```

4. Handle a report about gauge deletion

```
void onReportDeleted(in String UID,
        in String gaugeType, in String maugeName,
        in String modelType, in string modelName)
```

5. Handle a report about gauge configuration

```
void onReportConfigured(in String UID,
        in String gaugeType, in String gaugeName,
        in String modelType, in string modelName,
        in StringPairVector configParams)
```

---

[5] We anticipate that a separate proposal will elaborate on the services provided by gauge consumers for change management, system evaluation, etc.

4.3 Gauge Infrastructure Reference Implementation

4.3.1 The Layered Implementation Architecture

In this section, we describe an object-oriented implementation of the gauge infrastructure proposal. Because the main goal of gauge the infrastructure is to enable gauge reporting, and most of the time gauges are preoccupied with announcing information in an event-publishing way, an event bus is chosen as the communication mechanism for both publish-subscribe and RPC style connectors. On the event bus, caller and callee UIDs are included in RPC request and reply events to implement a peer-to-peer connection between a handle and its corresponding object.

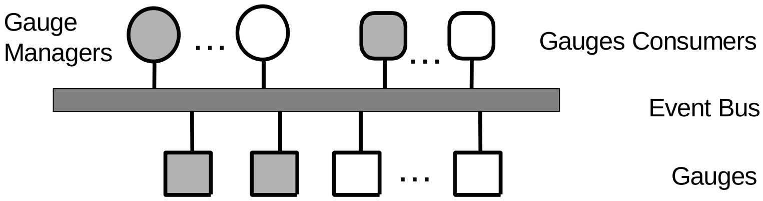

Figure 5. An event-based implementation of gauge Infrastructure

Figure 6 illustrates the overall structure of the reference implementation. The diagram is divided into three layers.

- The top layer includes three interfaces: *Gauge*, *GaugeMgr*, and *GaugeConsumer*, which need to be implemented by application programmers. Each interface includes functions that must be implemented by a gauge, a gauge manager or a gauge consumer, respectively.
    - The gauge consumer developer must implement the *GaugeConsumer* interface for each type of gauge consumer.
    - The gauge developer must implement the *Gauge* and *GaugeMgr* for each type of gauge.
- The middle layer includes three classes: *GaugeHandle*, *GaugeMgrHandle*, *GuageRptBus* and one *EventBus* interface. These are developed by gauge infrastructure developers.
    - *EventBus*[6] provides the common services required of an event bus connector, hiding the specific implementation.
    - *GaugeRptBus* provides the gauge infrastructure event service for gauges and gauge consumers. The *GaugeRptBus* is implemented on top of the *EventBus* class.
    - *GaugeHandle* is an implementation of a gauge handle built on top of the *EventBus* interface. It is generic and can be used for all gauge types. Gauge consumers can use *GaugeHandle* to operate a remote gauge, and gather value reports from it.
    - *GaugeMgrHandle* is an implementation of a gauge handle on top of the *GaugeMgrHandle* interface. It is generic and can be used for all gauge types.

---

[6] We use terminology that is consistent with Sienato describe the interface of *EventBus*.

Gauge consumers can use *GaugeMgrHandle* to operate a remote gauge manager, and gather value reports from it.

- The bottom layer includes possible communication mechanisms that can be used to implement the *EventBus*. The reference implementation uses the Siena Wide-Area Event Notification Service (developed at the University of Colorado, Boulder, because it provides platform-independent, internet-scale support for event publishing, subscription and filtering.

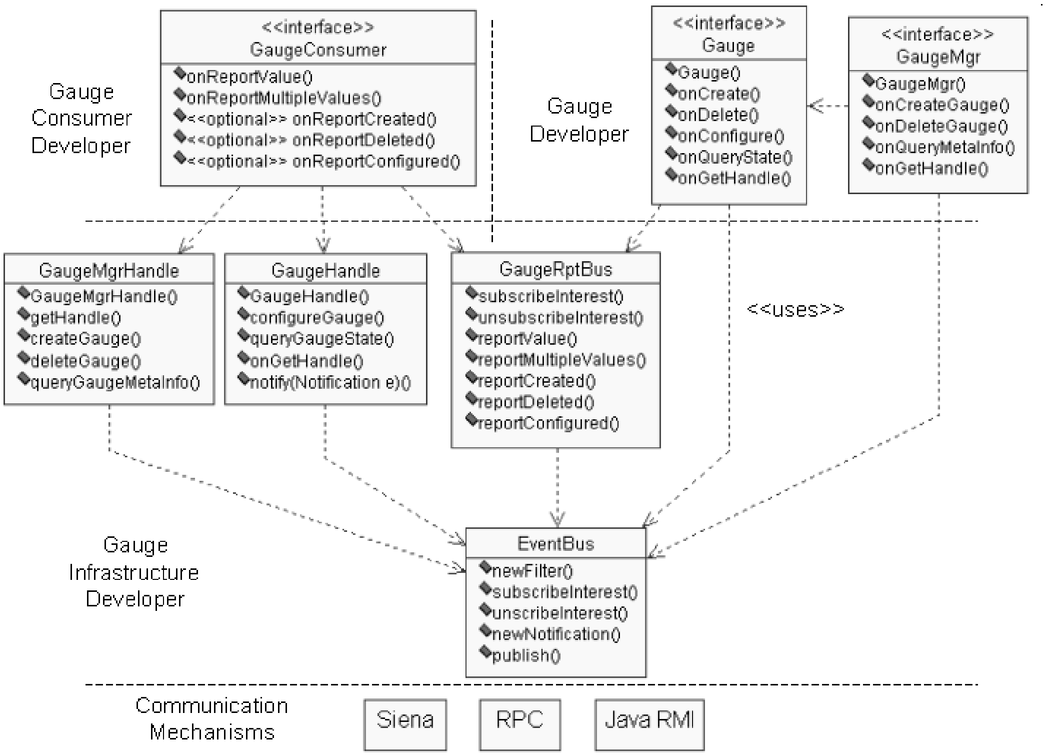

Figure 6. An OO Implementation of Gauge Infrastructure

The layered approach is a natural way to allocate responsibilities among different developers. The interface classes (*Gauge*, *GaugeMgr*, *GaugeConsumer*, and *EventBus*) encapsulate the differences and future changes of adjacent layer implementations. For example, we can change the underlying mechanism by only replacing the *EventBus* implementation, without affecting the existing gauge infrastructure.

4.3.2 Implementing Handles

The *GaugeHandle* and *Gauge* implementation together constitute the component "Gauge" in the logical architecture. The *GaugeMgrHandle* and *GaugeMgr* implementation together constitute the component "Gauge Manager" in the logical architecture.

As mentioned above, handles are implemented to emulate the RPCs on top of an event bus. To work correctly, each pair of <handle, class> must cooperate. They must share a common understanding of event format and mini-protocols (behaviors when seeing different notification sequences). The protocol part is actually described in Section 4.2 for each component. Currently the event format is not defined. XML is a reasonable candidate for representing events.

To implement handles, unique Ids are generated for each entity attached to the gauge reporting bus, and used as internal UIDs between interacting components. The handle object maintains the correspondence with a remote object using UIDs. Each function call involving a remote object will transparently use timeouts to detect abnormal situations and report errors.

4.4 Run Time Scenarios

Using the APIs defined above, a gauge consumer can create gauges and subscribe for values reported by them. Figure 7 shows two simple scenarios for gauge creation and gauge reporting.

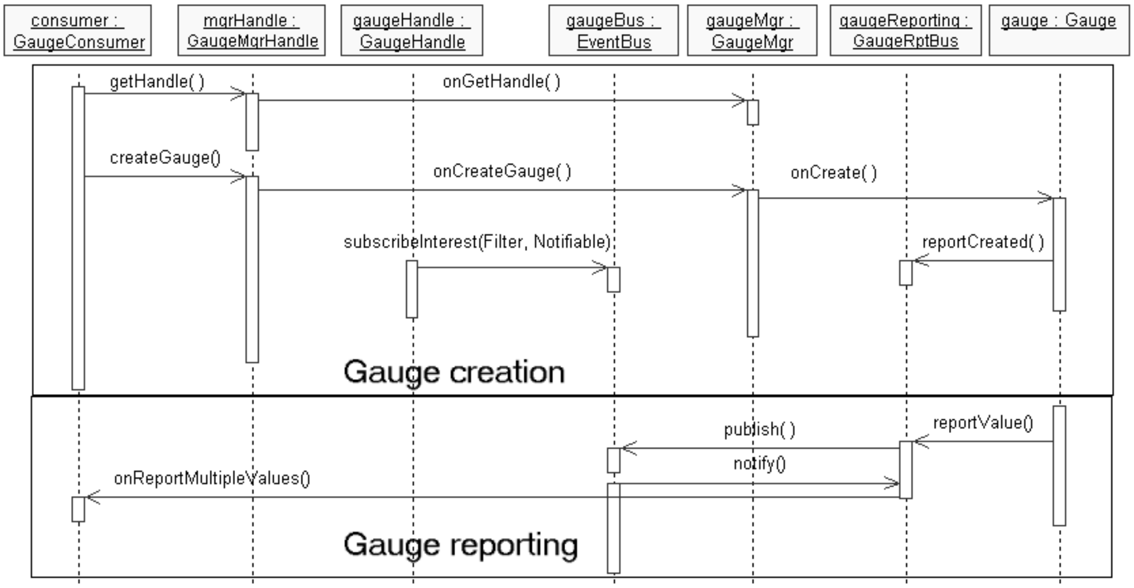

Figure 7. Gauge creation and gauge reporting scenarios

After implementation, the gauge developer will provide gauge managers to the run time infrastructure. Gauge consumers will be started and get the handle of a gauge manager that manages the types of gauges in which it is interested. The gauge consumer then uses the gauge manager handle to create gauges and return gauge handles as local references.

The sequence diagram in Figure 7 reflects the implementation outlined in Section 4.3. Upon creation, the gauge handle will automatically subscribe its interests in values reported by the corresponding gauge, unpack the event and return the values to its holder gauge consumer. *GaugeRptBus* object is added between the *Gauge* and *GaugeConsumer* to help the gauge publish its reports, and the gauge consumer to receive notifications.

4.5 Limitations of current implementation

The current implementation has the following limitations:

- The event subscription/unsubscription semantics is based on the model and implementation of Siena event notification service. The filtering constraints can only be combinations of certain *eventType*, *gaugeName*, *gaugeType*, *modelName*, and *modelType* values. We don't support the matching of event sequence patterns, which is supported by Siena.

- The current implementation assumes that one gauge manager implementation handles only one type of gauges.
- The current implementation assumes that the tuple of <*gaugeName*, *gaugeType*, *modelName*, *ModelType*> can be used to uniquely identify a gauge. This might not be true if multiple instances of the same system are running simultaneously.

5. Open Issues

5.1 Security Issues

Security can be an issue in some cases. Because one consumer's control of the gauge can have side effects that will influence other consumers, some gauges may want to require that only certain consumers can access the control interface. The gauge control interface may need to authenticate the initiator (taking the initiator as a parameter) to ensure that only authorized consumers can successfully perform such operations.

Another alternative would be for a consumer to control all gauges via their respective gauge managers. This would centralize the authentication, rather than having each gauge instance implement its own security protocol.

5.2 Gauge Identification

Currently the tuple of <*GaugeType, GaugeName, ModelType*, *ModelName*> is used by application programmers to identify a specific gauge. To maintain the uniqueness of gauge ID, we assume that there won't be simultaneous multiple incarnations of the same system (with the same *ModelName*). Combined with the gauge specification, such a tuple can be used by multiple consumers. This assumption is not always valid in reality.

Another way to uniquely identify an entity is to generate GUID at runtime. We choose to use UIDs as identifiers for entities attached to the gauge reporting bus. We also partly encapsulate the usage of UID by using handle, which can also be identified by a naming tuple.

5.3 Gauge Location

It's important to know where a newly created gauge should be placed: should it be near the target system, the owner consumer or the gauge manager? The gauge manager implements a gauge-specific placement policy. The consumer can also explicitly express its desire by passing location information in the setup parameters, and request the gauge factory to create new instance at a certain location. When a gauge needs to be placed on a host other than the gauge manager's own host, we also need to consider security issues.

5.4 Gauge Manager Location

Gauge manager location is another issue. When a gauge manager is needed, there are different methods to discover its location: (1) gauge managers advertise their existence and gauge consumers actively discover their locations, (2) the gauge consumer broadcasts the request and chooses from the responding gauge managers, (3) the connection between a gauge manager handler and its corresponding gauge manager is hardwired, and (4) develop a gauge manager discovery service to register and manage gauge managers. The location is encapsulated in the API by assuming that the connection between the gauge manager handle and the gauge manager is an implementation secret. The (2) approach is used in our implementation.

5.5 Race conditions

Race conditions need to be avoided in a gauge consumer implementation. For example, an event subscriber might miss some notification while the subscription is processed, so the subsequent actions that depend on such notifications should be programmed to assure their correctness.

Appendix A. The Acme Specification for Gauge Infrastructure

```
Family GaugeFamily = {
    port type gaugeLifeCycle = {};
    port type gaugeMetaQuery = {};
    port type publishReporting = {};
    port type onNotification = {};
    port type gaugeControl = {};
    port type gaugeQuery = {};
    port type requestGaugeControl = {};
    port type requestGaugeQuery = {};
    port type requestGaugeLifeCyle = {};
    port type requestGaugeMetaQuery = {};
    role type caller = {};
    role type callee = {};
    role type publisher = {};
    role type subscriber = {};

  Component Type Gauge = {
    port report : publishReporting = {};
    port control : gaugeControl = {};
    port query: gaugeQuery = {};
  };

  Component Type GaugeManager = {
    port lifeCycle : gaugeLifeCycle = {};
    port metaQuery : gaugeMetaQuery = {};
  };

  Component Type GaugeConsumer = {
    port onNot : onNotification = {};
  };

  Connector Type GaugeRPC = {
    role theCaller : caller = {};
    role theCallee : callee = {};
  };

  Connector Type GaugeReportingBus = {
    role pub : publisher = {};
    role sub : subscriber = {};
  };
};
```

```
System GaugeInfrastructure : GaugeFamily = {
  Component aGaugeMgr : GaugeManager = {
    port lifeCycle : gaugeLifeCycle = {};
    port metaQuery : gaugeMetaQuery = {};
  };
  component aGauge : Gauge = {
    port report : publishReporting = {};
    port control : gaugeControl = {};
    port query: gaugeQuery = {};
  };
  component aGaugeConsumer : GaugeConsumer = {
    port onNot : onNotification = {};
    port reqControl : requestGaugeControl = {};
    port reqQuery : requestGaugeQuery = {};
    port reqLifeCycle : requestGaugeLifeCyle = {};
    port reqMetaQuery : requestGaugeMetaQuery= {};
  };
  connector gaugeRptBus : GaugeReportingBus = {
    role pub : publisher = {};
    role sub : subscriber = {};
  };
  connector gControl : GaugeRPC = {
    role theCaller : caller = {};
    role theCallee : callee = {};
  };
  connector gQuery: GaugeRPC = {
    role theCaller : caller = {};
    role theCallee : callee = {};
  };
  connector gLifeCyl : GaugeRPC = {
    role theCaller : caller = {};
    role theCallee : callee = {};
  };
  connector metaQuery : GaugeRPC = {
    role theCaller : caller = {};
    role theCallee : callee = {};
  };
  Attachments {
    gaugeRptBus.pub to aGauge.report;
    gaugeRptBus.sub to aGaugeConsumer.onNot;
    gControl.theCaller to aGaugeConsumer.reqControl;
    gControl.theCallee to aGauge.control;
    gQuery.theCaller to aGaugeConsumer.reqQuery;
    gQuery.theCallee to aGauge.query;
    gLifeCyl.theCaller to aGaugeConsumer.reqLifeCycle;
    gLifeCyl.theCallee to aGaugeMgr.lifeCycle;
    metaQuery.theCaller to aGaugeConsumer.reqMetaQuery;
    metaQuery.theCallee to aGaugeMgr.metaQuery;
  }
}
```