To appear in the Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

Using Gauges for Architecture-Based Monitoring and Adaptation

David Garlan, Bradley Schmerl, and Jichuan Chang
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213
{garlan,schmerl,cjc}@cs.cmu.edu

Abstract.

An increasingly important requirement for complex systems is the capability to adapt at runtime in order to accommodate variable resources, system errors, and changing requirements. An essential aspect of adaptation is the ability to observe a system's runtime behavior and interpret those observations in terms that permit a high-level understanding of the system's status. In this paper we describe mechanisms for monitoring a running system, interpreting monitored data in terms of the system's architecture, and analyzing the system architecture to ascertain if the system's runtime behavior fits within the envelope of acceptable behavior. We illustrate the use of these mechanisms for the important special case of system performance monitoring.

1. Introduction.

An increasingly important requirement for systems is the ability to adapt to handle such things as resource variability, changing user needs, and system faults. Sometimes such adaptation can be user-assisted. For example, a user (or system administrator) might load new applications, change various system parameters, or reconfigure the load on a set of servers. But more and more, because of their inherent complexity and the need for rapid adaptation response, systems must handle their *own* adaptation, and do so while the system is running. An important engineering question then becomes how should one add such runtime adaptability mechanisms to complex software-based systems.

In the past, system adaptation has largely been handled internally to the application. For example, applications typically use generic mechanisms such as exception handling, or heartbeat mechanisms to trigger application-specific responses to an observed fault. For other aspects of adaptation, such as resource-based adaptation, systems typically wire in application-specific policies and mechanisms. For example, a video teleconferencing application may decide how to reduce its fidelity when transmission bandwidth is low using some combination of compression and reduced framesize and resolution.

While internal adaptation can certainly be made to work, it has a number of serious problems. First, when adaptation is intertwined with application code it is difficult and costly to make changes in adaptation policy and mechanism. Second, for similar reasons, it is hard to reuse adaptation mechanisms from one application to another. Third, it is difficult to reason about the correctness of a given adaptation mechanism, because one must also consider all of the application-specific functionality at the same time.

An alternative approach is to develop *externalized* adaptation mechanisms. For such systems, adaptation is handled outside of the application. Systems are monitored for various attributes, such as resource utilization, reliability, delivered quality of service. Based on that monitored information, external mechanisms decide whether the application must be reconfigured.

In contrast to the internal approach, externalized adaptation has many benefits: adaptation mechanisms can be more easily extended; they can be studied and reasoned about independently of the monitored applications; they can exploit shared monitoring and adaptation infrastructure.

One of the essential components of an externalized runtime adaptability mechanism is the ability to evaluate a system's status in terms that allow one to determine whether the system is operating within acceptable bounds. To do this effectively one must be able to collect information about the running system and interpret that information in the context of higher-level models that permit observations of system properties, such as its performance characteristics.

In this paper we address the problem of providing architecture-based performance monitoring for complex distributed systems. Specifically, we describe an ap-

proach in which we collect low-level performance information and then interpret that information in terms of architectural models, which can then be used as the basis for automated adaptation. We also describe the generic monitoring and interpretation infrastructure upon which our current implementation is based.

2. Dynamic Architecture-Based Adaptation.

Before describing our work on architecture-based monitoring, we start by describing our view of the necessary components for dynamic adaptation, the relevance of software architectures to this task, and a framework that is helping us to isolate and scope components of our research.

2.1. Required Pieces for Dynamic Adaptation.

Runtime system adaptation depends on three activities:

- **Monitoring**: There must be some way to observe the behavior of a running system. Without such information, it will not be possible to determine whether a change needs to be made. The infrastructure must support a wide variety of monitoring technologies. Given our desire to monitor a wide variety of applications, and that it is likely that these applications will be written in a variety of languages, different monitoring technologies are likely to be used. For example, it should be possible for the infrastructure to take advantage of monitoring technologies such as source code instrumentation to produce dynamic information, replacement of specific libraries with monitoring versions of those libraries, or services that give the state of the environment in which a program is running.
- **Interpretation**: The information that is being gathered must be aggregated and interpreted in the context of the higher-level system properties and features of one or more high-level models. For example, a monitored value that measures the bandwidth between two nodes on a network may not have an appropriate meaning until its effect on the system (for example, as expressed in the high-level model) can be determined (for example, in terms of completion or response time).
- **Reconfiguration**: Once a value is interpreted in the context of a model, it is possible to detect whether the system is running within acceptable parameters, or whether a change is needed. If a change is necessary, the target system is changed. In most cases, this will mean adapting the high-level model and ascertaining the correctness of the adaptation at that level before propagating the change to the implementation.

Given these three activities, the following questions need to be addressed by systems providing dynamic adaptation:

- Monitoring: What is the appropriate technology for providing monitoring of an application? How can support for a wide variety of monitoring techniques be provided? How can monitoring be added to existing systems in a way that minimizes the effect on the observed system?
- Interpretation: What kinds of high-level models are to be supported? How and when should this interpretation commence, and how often should it be made?
- Reconfiguration: What technologies should be used to determine if interpretations result in an incorrect system? How do we choose a particular reconfiguration that best addresses the errors detected in the system? How often, and under what conditions, should reconfiguration be applied? How is the reconfiguration accomplished in the running system?

2.2. Architectural Models.

One of the central issues in answering these questions is determining what kinds of models should be used to interpret observed behavior. In principle many kinds of models might be used, including behavioral, performance, structural, and timing models.

In our work we are focusing on the use of architectural models as the vehicle for monitoring, interpretation, and reconfiguration. An architectural model represents a system in terms of its principal run time parts ("components") and their pathways of communication ("connectors"). In addition, many architectural models include semantic detail that explains such things as expected properties of the components or connectors, constraints on topology or behavior, and allowable forms of evolution.

Architectural models are particularly useful for many kinds of automated system monitoring and adaptation. First, they provide a suitably high level of abstraction for interpreting low-level behavior in terms that an engineer or user can appreciate. For example, low-level details such as network routing pathways can be encapsulated as a single connector with values for throughput, latency, and congestion. These in turn can be interpreted in terms that are important to a user – such as information transmission time between two parts of a system. Second, they allow one to describe design constraints and expectations that can be evaluated against actual behavior. For example, when one associates explicit

protocols of interaction for a connector [1], tools can check that actual communication conforms to the expected protocol. Third, architectural models are often close enough to implementation structures that the system can automatically map architectural reconfiguration decisions to implementation reconfiguration commands.

2.3. Monitoring and Adaptation Framework.

Our architecture-based approach is based on the 3-layer view illustrated in Figure 1. Each conceptual layer consists of a set of representations, together with a manager responsible for monitoring and adapting the representations at that level. Inter-level interactions are facilitated by components, called *abstraction bridges*. This framework provides the context in which architectural models are used in dynamic adaptation.

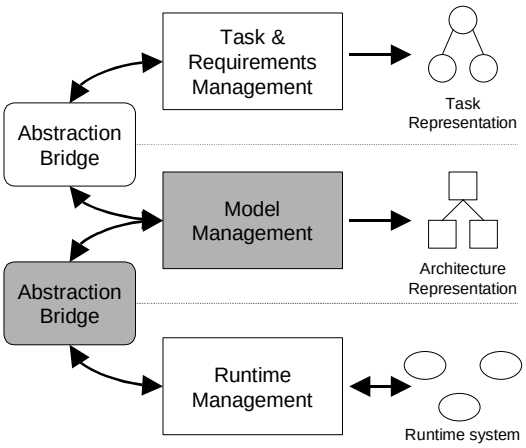

Figure 1. A Framework for Automated System Adaptation.

The lowest level is responsible for monitoring a system's runtime properties. It consists of the system itself, together with its operating environment (networks, processors, I/O devices, communications links, etc.). The *Runtime Management component* is responsible for monitoring actual system behavior and for causing changes to the system or its environment. Observed runtime information is propagated upwards through the abstraction bridge, which condenses, filters, and abstracts those observations in order to render that information in architecture-relevant terms.

The middle layer is responsible for interpreting observed system behavior in terms of higher-level, and more easily analyzed, properties. It forms the centerpiece of the approach, consisting of one or more architectural models of the system, together with *Model Managers* that determine when information from the lower level violates architectural design assumptions. In addition, the model management layer is responsible for determining repair strategies, and for communicating the corresponding control/reconfiguration operations to the lower level. For example, overload of a communication link (observed by the runtime manager) might violate minimal expectations of bandwidth of some connector in the architecture. An architecture manager, having detected the problem, would then change the connection mechanism to use another channel or communication protocol. Problems that cannot be handled by this layer are propagated up to the next layer. The top level is responsible for managing users' higher-level requirements. It maintains a representation of users' tasks, propagates system requirements down to the architectural modeling layer, and handles unrecoverable problems from below.

As a general framework, the scheme just described could be instantiated in many ways depending on the technical details of each part. At the lowest level, one could support many kinds of system observation: behavior (in the form of event traces), performance monitoring, intrusion detection, and real-time behavior are a few of the possibilities. At the middle layer, one could use any number of architectural models and architectural analysis capabilities. At the top level, one might have many ways of representing user tasks, including agents, process models, and workflow descriptions.

In our research we are instantiating this generic framework using the following technologies:

- **Level 1 – Performance Monitoring and Abstraction**: Our level-1 infrastructure currently supports monitoring of performance-oriented behavior. To do this we have adapted Remos [11], a Resource Monitoring System, that collects information on host and network loads and makes this information available through event announcement.
- **Level 2 – Architectural Modeling, Detection of Constraint Violations, and Automated Architectural Repair**: We use the Acme architecture description language [6] for representing the architectural model of the running system, together with design constraints that specify its expected behavior. We use AcmeStudio to make architectural models available at runtime, check for constraint violations, and provide an external API for modifying the architecture.
- **Level 3 – Task Management**: We plan to take advantage of existing strategies for representing tasks (such as workflow and process models, agents) and treat them as low-level task representations into which an even higher level of task description can be compiled.

The remainder of this paper focuses on the middle layer (the architectural models), and on the infrastructure for mapping low-level observations to architecturally relevant properties (the abstraction bridge). These are the shaded areas in Figure 1. We illustrate the use of this infrastructure and approach for the special, but important, case of architecture-based performance monitoring.

3. Gauges for Architecture-Based Monitoring.

In order to provide an abstraction bridge from system level values to observations in an architectural context, we have defined an infrastructure that uses probes and gauges.

The architecture of our runtime mechanisms is illustrated in Figure 2. 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 bus." At the second level a set of *gauges* 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 2 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; thus entities at the Model Management layer of the integration framework will form the set of gauge consumers.

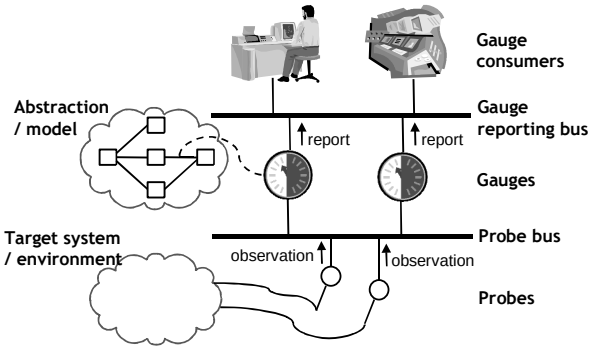

Figure 2. The Gauge Infrastructure.

As we will see, this infrastructure has a number of key features:

- Gauges are decoupled from the implemented system (by virtue of the probe layer.) This decoupling allows us to run gauges in a distributed fashion, so that they do not affect the performance of the system being gauged, and also gives the opportunity to use probe reports in a variety of ways for different models. For example, the same probe outputs can be used by gauges attached to other models or types of models, such as Meta-H or UML.
- Gauges can be mixed and matched, supporting interoperability between gauges that evaluate quite different properties and that are developed by different organizations.
- Gauges are insulated from lower-level transport mechanisms, enabling gauges to be deployed over both RPC-based channels and over publish-subscribe mechanisms.
- Gauges can be incorporated into architectural descriptions, enabling automatic generation and execution of gauges.

3.1. Performance Monitoring.

We have applied this general infrastructure to the specific case of performance monitoring. Our approach assumes that we use an architectural style that is amenable to performance monitoring and supports certain conventions about its mapping into an implementation. Specifically, we assume that connectors in the architecture map to TCP/IP connections in the implementation. This being the case, we can use probes in the implementation to measure and estimate bandwidth over TCP/IP connections (via Remos).

However, the information collected by probes at the implementation level is typically not at a sufficient level of abstraction required for architectural analysis. For example, we may be interested in the time taken to transfer certain amounts of information across a connector, or its latency, or other high-level observations of which bandwidth is merely a component. In such cases, gauges are used to collect information from probes and calculate observations that are relevant to the architectural model.

3.2. An Illustrative Example.

To illustrate how the infrastructure is used consider Figure 3, which presents a simple example of probing and gauging. Imagine that we have a target system consisting of a *sender* that sends files to a *server*. The architectural model of this system, represented at the top of the figure, consists of two components (the *sender* and the *server*) and one connector, *L*, representing the communication link between them. The implementation of this system consists of the programs comprising the sender and server (these could be further elaborated, but that is of no interest in this example), the actual network links between the machines on which the sender and server

are executing, and the set of files to be delivered. Note that a single high-level connector, *L*, can be realized by a complex network infrastructure, including many network segments, gateways, and routers. The user of this system requires that the set of files should reach the server within a certain deadline. Whether this deadline is being met by the running application depends on the size of the files to be transferred and the bandwidth available between the sender and the receiver. Thus, to ascertain the behavior of the system with respect to this performance attribute, we need to insert some probes and gauges.

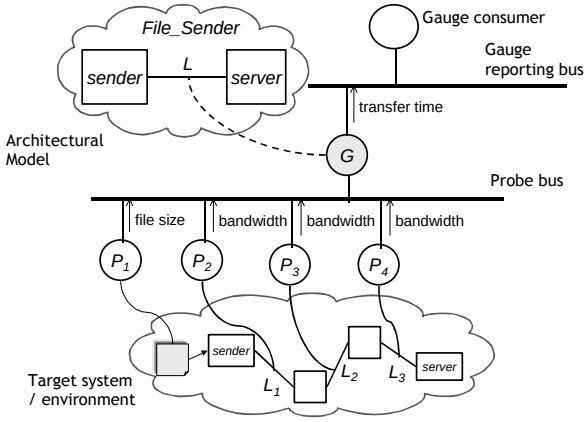

Figure 3. Gauge example.

Two types of probes are deployed in the target system. One type of probe monitors the environment, and reports the bandwidth of the various links between the machines of interest; these probes are represented by P_2, P_3, and P_4. The second type of probe, P_1, is inserted into the sender, and reports the size of the files being loaded into the system. Such a probe is realized by instrumenting the system call *fopen*. This probe information is not directly related to the performance attribute in the architectural model, which is in terms of transfer time between the *sender* and the *server*. To achieve this level of observation, a gauge is attached to the connector *L* in the architectural model. (Details of attachment are discussed Section 4.1). This gauge uses the probe values and calculates the estimated transfer time based on the file size and the available bandwidth. This value is then reported as the transfer time of that particular connector, to be consumed by a monitoring tool that will evaluate whether the deadline can be met.

The nature of probes, technologies for inserting them into systems, and how they report values is not discussed in detail in this paper. However, their context with respect to gauges is important in highlighting the difference between the low level, system observations and the high level, architectural observations that gauges produce.

3.3. 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 this notion of gauges and have been used to guide the design of the gauge specification and gauge APIs. These assumptions include:

1. *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 action 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.
2. *Different parties will develop different types of gauges.* 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.
3. *The set of gauge consumers can change dynamically.* In this way we can dynamically adapt our monitoring infrastructure to add new observational capabilities as needed.
4. *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.
5. *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 transfer time gauge of the example above interprets the physical

observations in terms of an abstract connector in the context of a specific architectural model.

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.

Given the diversity of gauges, created 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. Gauge developers can also use such a characterization as a functional specification around which to base their implementations, and by the gauge runtime 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.

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 transfer time gauge type, illustrated in Figure 3, 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 the transfer time value.

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

1. The name of the gauge type: for example, *XferTime_Gauge_T*;
2. The set of values reported by the gauge (specified using a name and a type): for example, the *XferTime_Gauge_T* reports one value, *xferTime* of type *float*;
3. Setup parameters (including name, type, and default value for each parameter): for example, the *XferTime_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 *XferTime_Gauge_T* has one configuration parameter *Sampling_Frequency*, which is of type *milliseconds* with a default value of *50*. The sets of configuration parameters and setup parameters are not necessarily disjoint. A default value should be provided for each configuration parameter that is not in the set of setup parameters.
5. Comments: these explain in more detail what a gauge does and how to interpret the values (the values' units, accuracies, etc.) and provide more detail about the functionality of the gauge.

To illustrate this definition, Table 1 describes a gauge type for the gauge *G* in Figure 3, that measures the transfer time value in milliseconds, represented as a floating point number.

Gauge Type	XferTime_Gauge_T
Reported Values	xferTime: 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.

How a given gauge type is instantiated is described in a gauge instance specification, which 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 3, which is of gauge type *XferTime_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 *File_Sender*, which is of model 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*: for example, the mapping for *G* is <*xferTime*, *L.xferTime*>;
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

when the gauge is created, the default value for 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: XferTime_Gauge_T
Model Name: Model Type	File_Sender : Acme
Mapping	<xferTime, L.xferTime>
Setup Values	Src_IP_Addr = L.src.IP1; Dst_IP_Addr = L.snk.IP2;
Configuration Values	Sample_Frequency = 100
Comments	G is associated with the L Connector of the system, File_Sender, defined as an Acme model.

Table 2. An Example of Gauge Instance Specification.

The above definition of gauges is very general and can be applied to a wide variety of monitoring needs, models, or modeling languages.

4. Implementation.

To this point we have described generally how gauges are specified and how they are used to monitor a system. Given this general infrastructure, we have experimented with a set of tools and techniques that allow monitoring in the context of Acme models. This section describes our implementation.

4.1. Attaching Gauges to Acme Descriptions.

As indicated earlier, gauges are used to interpret observations of the running system in the context of an architectural model. These observations form part of the semantics of the system and therefore should be mapped to the semantics of the architecture. Acme is a general-purpose architecture description language that is style-independent. Although particular styles can be defined in the Acme language, the building blocks of an architecture are generic components and connectors, with associated properties that do not have any inherent meaning. Styles are defined by specifying particular properties to be associated with particular types of components, and also in defining constraints that can be used to do some semantic analysis of the style. Furthermore, analysis tools can analyze certain properties in an architecture to arrive at some conclusion about the correctness of the architecture according to the analysis.

Because the semantics of an architecture are captured in the property mechanism of Acme, gauges are attached to Acme properties. The meaning of this is that the value(s) reported by a gauge are actually values of the properties to which they are attached. In this way, architecture-based analysis tools can observe these changing properties. For example, design constraints over the properties in an architecture can be re-evaluated when a property value changes dynamically as reported by a gauge. This allows any other tools that analyzes Acme properties to be used dynamically. Attaching gauges to properties also means that tools that currently work with Acme descriptions need not change when gauges are added.

```
system file_sender : GaugedClientServerFam = {
  property Ggauge : XferTime_Gauge_spec = [
    name = "G";
    gaugeType = XferTime_Gauge_T;
    setup = [   Src_IP_Addr = "barossa.cs.cmu.edu";
                Dst_IP_Addr = "hunter.cs.cmu.edu"
    ];
    configuration = [Sampling_Frequency = 100.0];
  ] <<isGauge : boolean = true;>>;...
  connector L = {
    role src;
    role snk;
    property xferTime : float
      <<gauged : boolean = true;
        gauge = [
          name = "G";
          value = xferTime
        ];
      >>;
    };
};
```

Figure 4. A Gauge Attached to an Acme connector.

To attach a gauge to an Acme property, it is first necessary to define the gauge as a property of the system. The property *Ggauge* in the *file_sender* system in Figure 4 defines a gauge and gives it a name, a type, and defines the setup and configuration parameters. The type of this property (*XferTime_Gauge_spec*) is defined in the family of which the system is an instance. Tools can determine that this is a gauge by looking at the meta-property *isGauge:* if it is defined, then that property is intended to be a gauge.[1] Figure 4 shows an Acme

[1] Meta-properties are currently used in Acme to assign details like default values or units of measure and enclosed by << >>.

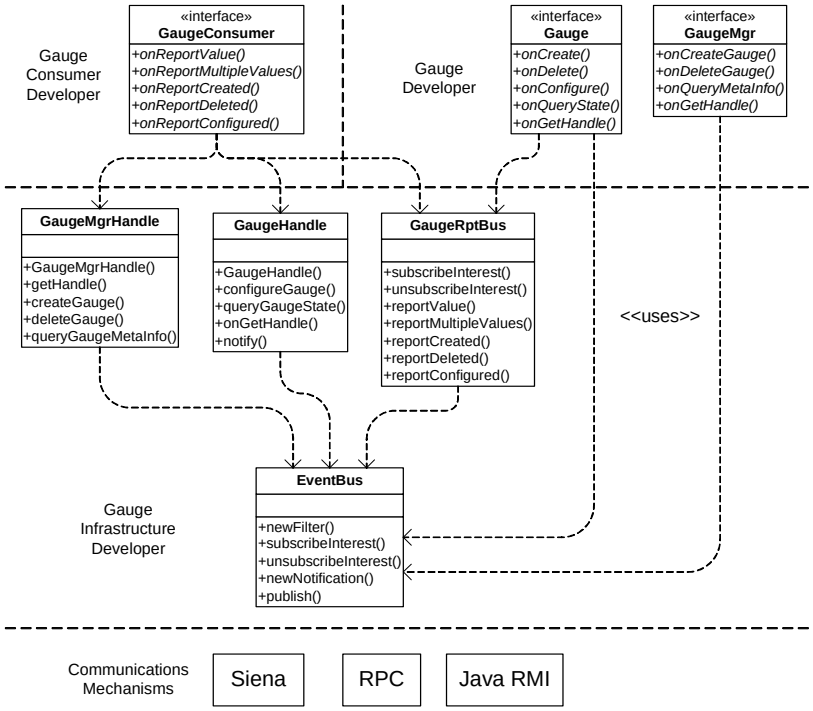

Figure 5. Gauge Infrastructure Implementation.

description of the connector *L* from Figure 3. The fact that a property is a gauged value (and therefore its value is assigned at runtime) is set by having the meta-property *gauged* associated with the property. The next meta-property (*gauge*) references the attached gauge and defines which gauge value is mapped to this particular property. The Acme gauge specification in Figure 4 corresponds to the gauge instance specification in Table 2. Each gauge type has a corresponding Acme record type. These records can be generated automatically from the gauge type specification.

4.2. Tool Support for Gauges.

Based on the definition of gauges given above, we have developed an implementation of the gauge infrastructure that provides a set of Java classes and interfaces, and one of several communication substrates through which events are communicated between gauges and their consumers. The class hierarchy for this implementation is shown in Figure 5. The classes provided by the infrastructure are shown in the middle of the figure; the interfaces that need to be implemented for particular gauges or gauge consumers are at the top of the figure. This implementation hides the communication mechanism used to send the events. In fact, we have one implementation that uses the Siena wide-area event notification system [3], and another that utilizes Java RMI to transport events – in either case, the code that the Gauge Developer or Gauge Consumer Developer has to write is exactly the same, allowing portability across communication mechanisms.

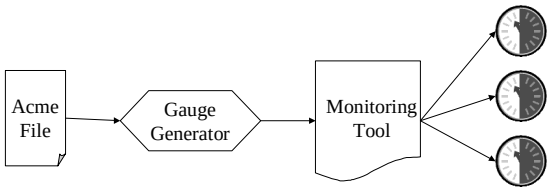

Figure 6. Generating Gauges from Acme Descriptions.

An Acme description with a set of attached gauges can be used to generate the gauge instances so that consumers can listen to those messages. We have written a tool that does this, and also generates the necessary code to connect with our design environment, AcmeStudio, which can be used to display the gauge outputs dynamically. Figure 6 shows the process by which this is achieved. The *Gauge Generator* takes the Acme file and produces a *Monitoring Tool*. This tool, when executed, will create and configure the gauge instances (by connecting with the appropriate gauge managers), and uses

the COM interface of AcmeStudio to load the Acme description, start listening for gauge values, and propagate these values to AcmeStudio.

Figure 7 illustrates an instance of the gauge infrastructure with associated tools after the tool described above is run. The tool first creates the gauge, in the center of the figure, and sets it up so that it is reporting its value in the context of the system's architecture (it is attached to the *xferTime* property of the connector *L*). Once this gauge is created, it activates various probes in the target system. Remos, a network service that gives applications access to the available bandwidth, is used to report the bandwidth between the sender and the server. The Remos probe is focused so that it reports information for the network between the hosts described in the *ip* property of the roles in the connector *L*. Currently, the filesize probe is hardwired into the sender; more disciplined approaches to deploying and activating probes are being developed by others, and will be used in the implementation of the gauges as they become available. The tool also connects to AcmeStudio, via a COM interface, and loads the description to which the gauge is attached. As it receives reports from the gauge, it changes the associated property via this interface, and AcmeStudio displays the value changing as it is reported.

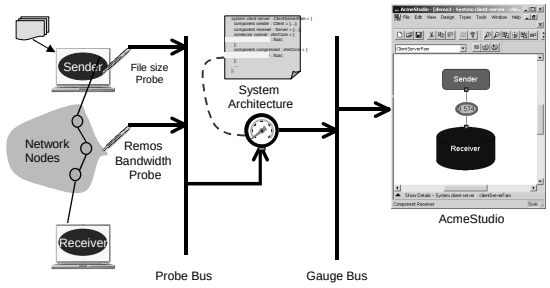

Figure 7. Implementation of Monitoring described in Figure 3.

5. Related Work.

Considerable research has been done in the area of dynamic adaptation of systems, for both internal and external adaptation. In the internal realm, there are a multitude of programming languages and libraries that provide dynamic linking and binding mechanisms (e.g., [4, 9, 10, 13]) Systems of this kind allow dynamic change to be programmed on a per-system basis, and do not provide external, reusable mechanisms that can be added to systems in a disciplined manner *per se*, as with our approach.

Many researchers have tried to address monitoring and distributed debugging as an external facility of the environment. However, this is a hard problem to solve in general. Our approach is to narrow the problem by (a) using an architectural model as the basis for analyzing and interpreting monitored values; (b) providing collections of property-specific gauges that can be attached to these architectures (in this paper, we discuss performance monitoring); and (c) assuming certain regularities in implementation that allow us to exploit generic performance monitoring network probes (in this paper, we use Remos) and map from the architectural model to the running system. By scoping the problem in this way, we are able to make tractable many of the thorny problems of abstraction and interpretation of monitored data.

Several researchers have investigated the use of software architecture in supporting adaptation. Oreizy et al. [16, 17, 18] detail a framework for architecture-based adaptation, in which change is mediated by an architecture evolution manager. This entity corresponds to the abstraction bridge in Figure 1. System adaptation in their research leverages the use of specific architectural styles – publish-subscribe as in C2 [21], or data flow as in Weaves [7, 8]. Similarly, Darwin [15] (and its predecessor, Conic [14]) provides a "Configuration Manager" for making dynamic changes, and uses a style based on bi-directional communication links. These styles provide loose coupling between components making it possible to remove and insert new components and connectors. They also lead to stereotyped implementations so that architectural changes can be easily mapped to corresponding system changes.

In contrast, our approach is not tied to specific styles. This has the benefit that adaptation infrastructure and monitoring capabilities can be applied to a variety of styles. For example, our performance monitoring infrastructure can be applied to any style, provided the connectors in the architecture can be mapped to well-defined TCP/IP network channels. On the other hand, the flexibility in our approach comes at a cost: it may be difficult to bridge the gap between implementation and architecture, and implementations may not be inherently capable of certain kinds of adaptation that we would like to perform. The first of these limitations is addressed structurally in our approach by the two-level monitoring scheme in which probes are tied to specific implementations and gauges are responsible for bridging the gap between implementation and architecture.

Furthermore, because we are using a language in which architectural design constraints are easily expressed [12], we can check the conformance of observations of implementations to architectural design assumptions. Also, work done in DCDL [19, 20] for applying

consistency constraints specified in a first-order predicate logic language to determine consistent configurations of dynamically adapting systems is applicable to our use of Armani in analyzing correct software architecture. DCDL uses these consistency rules to determine the set of existing versions of components will result in consistent configurations.

6. Conclusions and Future Work.

In this paper, we have introduced a framework and an implementation that provides a general mechanism for adding monitoring and interpretation capabilities that takes advantage of probes to observe the performance of executing programs, gauges to interpret those observations, and a strategy that allows these gauges to be attached to properties in an Acme description. This framework and implementation has the advantage that it is:

- Distributed, so that monitoring and assessment of a running application can occur without needing to consume extensive resources on the machine on which the application being monitored is executing;
- General, in that the monitoring capability is not restricted to a particular architectural or implementation style.

Specifically, we have described our approach to providing externalized dynamic adaptation, and our design and implementation of the monitoring component of this approach. Although we have illustrated our approach in the context of a particular quality attribute (performance), we believe the monitoring component is general enough to allow us to use many "probe" technologies, that observe the state of the implemented system. The values of these low-level observations can be interpreted at the architectural level by attaching gauges to architectural properties. This approach satisfies the requirements that such a technology be general and distributed. We view this work as a first step for achieving the overall framework described earlier. In our ongoing research we plan to extend our results and evaluate the current design. Our plans for evaluation include:

- Applying different kinds of probing technologies in a principled manner. The Remos probe that we have used to observe low-level information is a probe that monitors the network environment of a distributed application. It does not need to be inserted into an application, and so is non-intrusive. Currently, the other probes described in this paper, is handcoded into the application. We plan to coordinate with other researchers developing different types of probe technologies, to evaluate the generality of our gauges with respect to probes.
- Evaluating the performance overhead and synchronization issues with our performance monitoring technology. Our technology, while measuring the available bandwidth, also affects the bandwidth because it may communicate the report along the very channels it is measuring. We have attempted to minimize this by executing the Remos collector and probe on a separate machine to the ones executing the application.
- A further problem is the time it takes to report and react to values being communicated. For example, if reporting is slow to reach the architectural level, then, by the time a change is required, the conditions in the environment may have change sufficiently to warrant a different change or require no change at all. The effects of this problem, and ways to alleviate it, are areas of further study.

While we have described in some detail the monitoring component, the harder tasks, that of deciding on a change and actually conducting the change, are yet to be done. We plan to take advantage of our previous work in architectural analysis to guide us in how to analyze reports; for this, we will initially use Armani constraints to provide feedback on whether a monitored architecture requires a change. Our next step is to develop a formalism for taking detected errors and associating them with particular changes that will correct the errors – a component we call repair strategies. We also plan to use and extend the work on Aesop [5] to guide us in taking changes at the architectural level and reflecting them in the implementation.

Acknowledgements

DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616, supports this work. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA. We would like to thank the members of the ABLE group.

References

[1] "A Formal Basis for Architectural Connection," R. J. Allen and D. Garlan, in ACM Transactions on Software Engineering, July 1997.

[2] "Specifying and Analyzing Dynamic Software Architectures," R. J. Allen, R. Douence, and D. Garlan, in *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering* (FASE '98), March 1998.

[3] "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service," A. Carzaniga, D.S. Rosenblum, and A.L. Wolf in *Proceedings of the Nine-*

teenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR. July, 2000.

[4] "Dynamic Binding in Strongly Typed Programming Languages," R. E. Gantenbein, *Journal of Systems and Software* **14**(1):31—38, 1991.

[5] "Exploiting Style in Architectural Design Environments," D. Garlan, R. Allen, and J. Ockerbloom, *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng*, December 1994.

[6] "Acme: Architectural Description of Component-Based Systems," D. Garlan, R. T. Monroe, and D. Wile, in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman (eds), Cambridge University Press, 2000, pp. 47-68.

[7] "Using Weaves for Software Construction and Analysis," M.M. Gorlick and R.R. Razouk, in *Proceedings of the 13th International Conference on Software Engineering*, IEEE Computer Society Press, May 1991.

[8] "Visual Programming-in-the-Large versus Programmming-in-the-Small," M.M. Gorlick and A. Quilici, in *Proceedings of the IEEE Symposium on Visual Languages*, IEEE Computer Society Press, October 1994.

[9] "The Java Language Environment: A White Paper," J. Gosling and H. McGilton, Sun Microsystems Computer Company, Mountain View, California, May 1996. Available at http://java.sun.com/docs/white/langenv/.

[10] "An Approach to Genuine Dynamic Linking," W. W. Ho and R. A. Olsson, *Software – Practice and Experience* **21**(4):375—390, 1991.

[11] "A Resource Query Interface for Network-Aware Applications," B. Lowekamp, N. Miller, D. Sutherland, D. Gross, P. Steenkiste, and J. Subhlok, in *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing* (Chicago, IL), July 1998.

[12] "Capturing Software Architecture Design Expertise With Armani," R. Monroe, CMU School of Computer Science Technical Report CMU-CS-98-163. Version 2.3, revised January 2001.

[13] "Exploiting Persistent Linkage in Software Engineering Environments," R. Morrison, R. C. H. Connor, Q. I. Cutts, V. S. Dunstan, and G. N. C. Kirby, *The Computer Journal* **38**(1):1—16, 1995.

[14] "Constructing Distributed Systems in Conic," J. Magee, J. Kramer, and M. Sloman, *IEEE Transactions of Software Engineering* **15**(6), 1989.

[15] "Specifying Distributed Software Architectures," J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, in *Proceedings of 5th European Software Engineering Conference* (ESEC '95), Sitges, September 1995. Also published as Lecture Notes in Computer Science 989, (Springer-Verlag), 1995, pp. 137-153.

[16] "Architecture-Based Runtime Software Evolution," p. Oriezy, N. Medvidovic, and R.N. Taylor, in the *Proceedings of the International Conference on Software Engineering 1998* (ICSE'98). Kyoto, Japan, April 1998, pp. 11—15.

[17] "On the Role of Software Architectures in Runtime System Reconfiguration," P. Oreizy and R. N. Taylor, in *Proceedings of the International Conference on Configurable Distributed Systems* (ICCDS 4). Annapolis, Maryland, May 4-6, 1998.

[18] "An Architecture-Based Approach to Self-Adaptive Software," P. Oreizy, M. M. Gorlick, R. N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 54—62.

[19] "Consistency Issues in Partially Bound Dynamically Composed Systems," B. R. Schmerl and C. D. Marlin, in *Proceedings of the 1996 Australian Software Engineering Conference* (Melbourne, Australia), pp 183-192. IEEE Computer Society Press, Los Alimitos, California. July 1996.

[20] "Designing Configuration Management Tools for Dynamically Composed Systems," B. R. Schmerl. Ph.D. Thesis, Department of Computer Science, Flinders University of South Australia, December 1997.

[21] "A Component- and Message-Based Architectural Style for GUI software," R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oriezy, and D.L. Dubrow, in *IEEE Transactions on Software Engineering*, pp. 390-406, June 1996.