Appears in *Architecting Dependable Systems*, de Lemos, Gacek, Romanovsky (eds)
2003, © Springer-Verlag.

Increasing System Dependability through Architecture-based Self-repair

David Garlan, Shang-Wen Cheng, Bradley Schmerl

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
{garlan, zensoul, schmerl}@cs.cmu.edu

Abstract. One increasingly important technique for improving system dependability is to provide mechanisms for a system to adapt at run time in order to accommodate varying resources, system errors, and changing requirements. For such "self-repairing" systems one of the hard problems is determining when a change is needed, and knowing what kind of adaptation is required. In this paper we describe a partial solution in which stylized architectural design models are maintained at run time as a vehicle for automatically monitoring system behavior, for detecting when that behavior falls outside of acceptable ranges, and for deciding on a high-level repair strategy. The main innovative feature of the approach is the ability to specialize a generic run time adaptation framework to support particular architectural styles and properties of interest. Specifically, a formal description of an architectural style defines for a family of related systems the conditions under which adaptation should be considered, provides an analytic basis for detecting anomalies, and serves as a basis for developing sound repair strategies.

1. Introduction

One increasingly important technique for improving software-based system integrity is providing systems with the ability to adapt themselves at run time to handle such things as resource variability, changing user needs, and system faults. In the past, systems that supported such self-adaptation were rare, confined mostly to domains like telecommunications switches or deep space control software, where taking a system down for upgrades was not an option, and where human intervention was not always feasible. However, today more and more systems have this requirement, including e-commerce systems and mobile embedded systems. Such systems must continue to run with only minimal human oversight, and cope with variable resources (bandwidth, server availability, etc.), system faults (servers and networks going down, failure of external components, etc.), and changing user priorities (high-fidelity video streams at one moment, low fidelity at another, etc.).

Traditionally system self-repair has been handled within the application, and at the code level. For example, applications typically use generic mechanisms such as exception handling or timeouts to trigger application-specific responses to an observed

fault or system anomaly. Such mechanisms have the attraction that they can trap an error at the moment of detection, and are well-supported by modern programming languages (e.g., Java exceptions) and run time libraries (e.g., timeouts for RPC). However, they suffer from the problem that it can be difficult to determine what the true source of the problem is, and hence what kind of remedial action is required. Moreover, while they can trap errors, they are not well-suited to recognizing "softer" system anomalies, such as gradual degradation of performance over some communication path, or transient failures of a server.

Recently a number of researchers have proposed an alternative approach in which system models – and in particular, architectural models – are maintained at run time and used as a basis for system reconfiguration and repair [32]. Architecture-based adaptation has a number of nice properties: As an abstract model, an architecture can provide a global perspective on the system, enabling high-level interpretation of system problems. This in turn allows one to better identify the source of some problem. Moreover, architectural models can make "integrity" constraints explicit, helping to ensure the validity of any system change.

A key issue in making this approach work is the choice of architectural style used to represent a system.[1] Previous work in this area has focused on the use of specific styles (together with their associated description languages and toolsets) to provide intrinsically modifiable architectures. Taylor et al. use hierarchical publish-subscribe via C2 [31, 36]; Gorlick et al. use a dataflow style via Weaves [14]; and Magee et al. use bi-directional communication links via Darwin [22].

The specialization to particular styles has the benefit of providing strong support for adapting systems built in those styles. However, it has the disadvantage that a particular style may not be appropriate for an existing implementation base, or it may not expose the kinds of properties that are relevant to adaptation. For example, different styles may be appropriate depending on whether one is using existing client-server middleware, Enterprise JavaBeans (EJB), or some other implementation base. Moreover, different styles may be useful depending on whether adaptation should be based on issues of performance, reliability, or security.

In this paper we show how to generalize architecture-based adaptation by making the choice of architectural style an explicit design parameter in the framework. This added flexibility allows system designers to pick an appropriate architectural style in order to expose properties of interest, provide analytic leverage, and map cleanly to existing implementations and middleware.

The key technical idea is to make architectural style a first class run time entity. As we will show, formalized architectural styles augmented with certain run time mechanisms provide a number of important capabilities for run time adaptation: (1) they define a set of formal constraints that allow one to detect system anomalies; (2) they are often associated with analytical methods that suggest appropriate repair strategies; (3) the allow one to link stylistic constraints with repair rules whose soundness is based on corresponding (style-specific) analytical methods; (4) they provide a set of operators for making high-level changes to the architecture; (5) they prescribe what aspects of a system need to be monitored.

---

[1] By "architectural style" we mean a vocabulary of component types and their interconnections, together with constraints on how that vocabulary is used.

In the remainder of this paper we detail the approach, focusing primarily on the role of architectural styles to interpret system behavior, identify problems, and suggest remediation. To illustrate the ideas we describe how the techniques have been applied to self-repair of an important class of web-based client-server systems, based on monitoring of performance-related behavior. As we will show, the selection of an appropriate architectural style for this domain permits the application of queuing-theoretic analysis to motivate and justify a set of repair strategies triggered by detection of architectural constraint violations.

2. Related Work

Considerable research has been done in the area of dynamic adaptation at an implementation level. There are a multitude of programming languages and libraries that provide dynamic linking and binding mechanisms, as well as exception handling capabilities (e.g., [8, 16, 18, 27]). Systems of this kind allow system self-repair to be programmed on a per-system basis, but do not provide external, reusable mechanisms that can be added to systems in a disciplined manner *per se*, as with an architecture-driven approach.

Our work is also related to distributed debugging systems, insofar as remotely monitoring a running system to locate problems [15]. However, those systems have focused on user-mediated monitoring, whereas our research is primarily concerned with automated monitoring and reconfiguration. Adaptive or reflective middleware attempts to provide some automated support for adaptation of distributed applications, through shared infrastructure for component integration. An adaptive middleware supports inspection and modification of its internal state, and enables high-level abstraction for greater ease in controlling the lower-level services provided by the middleware [1, 20]. This work is similar to ours in that the middleware maintains an explicit representation of its internal structure and uses that model to adjust its properties. While adaptive middleware technology gives an application greater flexibility to adapt to changing requirements and environments, it is focused at adapting shared infrastructure. Our work in contrast also allows adaptation of the applications running on *top* of such infrastructure.

The most closely related research is the work on architecture-based adaptation, mentioned earlier. As we noted, the primary difference between our work and earlier research in this area is the decoupling of style from the adaptive system infrastructure so that developers have the flexibility to pair an appropriate style to a system based on its implementation and the system attributes that should drive adaptation. To accomplish this we have to introduce some new mechanisms to allow "run time" styles to be treated as a design parameter in the run time adaptation infrastructure. Specifically, we must show how styles can be used to detect problems and trigger repairs. We must also provide mechanisms that bridge the gap between an architectural model and an implementation – both for monitoring and for effecting system changes. In contrast, for systems in which specific styles are built-in (as with [14, 35]) this is less of an issue because architectures are closely coupled to their implementations *by construction*.

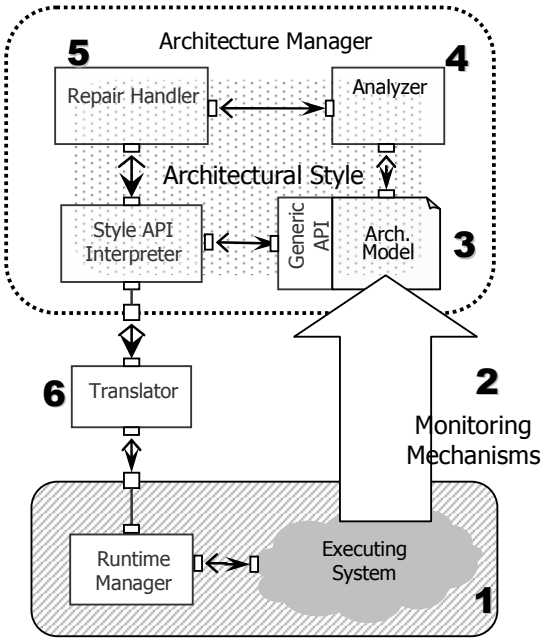

Fig. 1. Adaptation Framework

Finally, there has been some work on formally characterizing architectural styles, and using them as a basis for system analysis [12, 35]. Our research extends this by showing how to turn "style as a design time artifact" into "style as a run time artifact". As we will see, this change requires two significant additions to the usual notion of style as a set of types and constraints: (1) style-specific repair rules, and (2) style-specific change operators. Some other efforts in this area have investigated formal foundations for dynamic architectures in terms of graph grammars and protocols, but have not attempted to use those formal descriptions as part of the run time adaptation infrastructure [3, 24, 40].

3. Overview of Approach

Our starting point is an architecture-based approach to self-adaptation, similar to [32] (as illustrated in Figure 1): In a nutshell, an executing system (1) is monitored to observe its run time behavior; (2) Monitored values are abstracted and related to architectural properties of an architectural model; (3) Changing properties of the architectural model trigger architectural analysis to determine whether the system is operating within an envelope of acceptable ranges; (4) Unacceptable operation causes repairs, which (5) adapt the architecture; (6) Architectural changes are propagated to the running system.

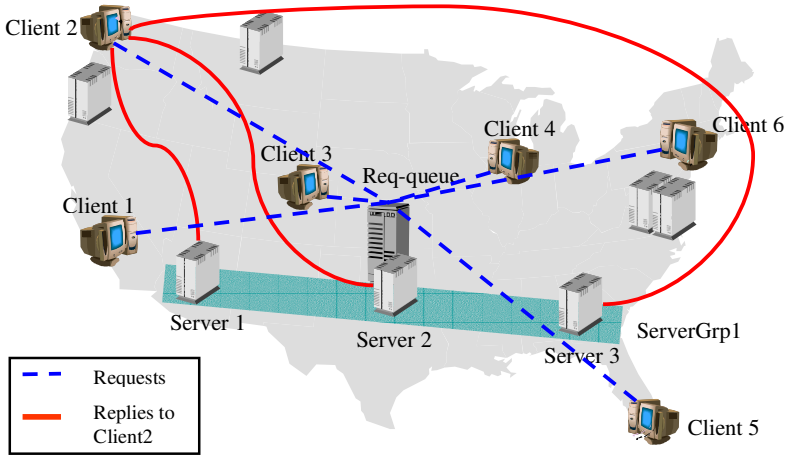

Fig. 2. Deployment Architecture of Example System.

The key new feature in this framework is the use of style as a first class entity that allows one to tailor the framework to the application domain, and determines the actual behavior of each of the parts. Specifically, style is used to determine (a) what properties of the executing system should be monitored, (b) what constraints need to be evaluated, (c) what to do when constraints are violated, and (d) how to carry out repair in terms of high-level architectural operators. In addition we need to introduce a style-specific translation component to manage the transactional nature of repair and map high-level architecture operations into lower-level system operations.

To illustrate how the approach works, consider a common class of web-based client server applications that are based on an architecture in which web clients access web resources by making requests to one of several geographically distributed server groups (see Figure 2). Each server group consists of a set of replicated servers, and maintains a queue of requests, which are handled in FIFO order by the servers in the server group. Individual servers send their results directly to the requesting client.

The organization that manages the overall web service infrastructure wants to make sure that two inter-related system qualities are maintained. First, to guarantee quality of service for the customer, the request-response latency for clients must be under a certain threshold (e.g., 2 seconds). Second, to keep costs down, the set of currently active servers should be kept as loaded as possible, subject to the first constraint.

Since access loads in such a system will naturally change over time, the system has two built-in low-level adaptation mechanisms. First, we can activate a new server in a server group or deactivate an existing server. Second, we can cause a client to shift its communication path from one server group to another.

The challenge is to engineer things so that the system adapts appropriately at run time. Using the framework described above, here is how we would accomplish this.

First, given the nature of the implementation, we decide to choose an architectural style based on client-server in which we have clients, server groups, and individual servers, together with the appropriate client-server connectors (see Figure 3). Next, because performance is the key quality attribute of concern, we adapt that style so that it captures performance-related properties and makes explicit constraints about acceptable performance (see Figure 4). Here, client-server latency and server load are the key properties, and the constraints are derived from the two desiderata listed above. Furthermore, because of the nature of communication we are able to pick a style for which formal performance analyses exist – in this case M/M/m-based queuing theory.

To make the style useful as a run time artifact we now augment the style with two specifications: (a) a set of style-specific architectural operators, and (b) a collection of repair strategies written in terms of these operators and associated with the style's constraints. The operators and repair strategies are chosen based on an examination of the analytical equations, which formally identify how the architecture must change in order to affect certain parameters (like latency and load).

There are now only two remaining problems. First, we must get information out of the running system. To do this we employ low-level monitoring mechanisms that instrument various aspects of the executing system. We can use existing off-the-shelf performance-oriented "system probes," which we detail later. To bridge the gap between low-level monitored events and architectural properties we use a system of adapters, called "gauges," which aggregate low-level monitored information and relate it to the architectural model. For example, we have to aggregate various measurements of the round-trip time for a request and the amount of information transferred to produce bandwidth measurements at the architectural level.

The second problem is to translate architectural repairs into actual system changes. To do this we write a simple table-driven translator that can interpret architectural repair operators in terms of the lower level system modifications that we listed earlier. In the running system the monitoring mechanisms update architectural properties, causing reevaluation of constraints. Violated constraints (high client-server latencies, or low server loads) trigger repairs, which are carried out on the architectural model, and translated into corresponding actions on the system itself (adding or removing servers, and changing communication channels). The existence of an analytic model for performance (M/M/m queuing theory) helps guarantee that the specific modification operators for this style are sound. Moreover, the matching of the style to the existing system infrastructure helps guarantee that relevant information can be extracted, and that architectural changes can be propagated into the running system.

4. Style-based Adaptation

In this section, we discuss in more detail each aspect of the architectural adaptation framework. We begin with an introduction on software architecture and architectural styles, and proceed to discuss the changes to these ideas necessary to make them available and useful for dynamic adaptation. We then discuss the techniques for ob-

serving and affecting the running system. In the next section, we give an example of the entire architectural style based on the example introduced in Section 3.

4.1 Architectural Models and Styles

The centerpiece of our approach is the use of stylized architectural models. Although there are many modeling languages and representation schemes for architecture, we adopt a simple approach in which an architectural model is represented as an annotated, hierarchical graph.[2] Nodes in the graph are *components*, which represent the principal computational elements and data stores of the system. Arcs are *connectors*, which represent the pathways of interaction between the components. Components and connectors have explicit interfaces (termed *ports* and *roles,* respectively). To support various levels of abstraction and encapsulation, we allow components and connectors to be defined by more detailed architectural descriptions, which we call *representations.*

To account for semantic properties of the architecture we allow elements in the graph to be annotated with extensible property lists. Properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). Properties associated with a component might define its core functionality, performance attributes (e.g., average time to process a request, load, etc.), or its reliability.

Representing an architecture as an arbitrary graph of generic components and connectors has the advantage of being extremely general and open ended. However, in practice there are a number of benefits to constraining the design space for architectures by associating a *style* with the architecture. An architectural style typically defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed .

Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [12, 35, 38]. Moreover, the notion of style often maps well to widely-used component integration infrastructures (such as EJB, HLA, CORBA), which prescribe the kinds of components allowed and the kinds of interactions that may take place between them.

As a result, a number of Architecture Description Languages (ADLs) and their toolsets have been created to support system development and execution for specific styles. For example, C2 [36] supports a style based on hierarchical publish-subscribe; Wright [2, 3] supports a style based on formal specification of connector protocols; MetaH [38] supports a style based on real-time avionics control components.

In our research we adopt the view that while choice of style is critical to supporting system design, execution, and evolution, different styles will be appropriate for different systems. For example, a client-server system, such as the one in our example, will most naturally be represented using a client-server style. In contrast, a signal processing system would probably adopt a dataflow-oriented pipe-filter style. While one might *encode* these systems in some other style, the mapping to the actual system

[2] This is the core architectural representation scheme adopted by a number of ADLs, including Acme [12], xArch [8], xADL [9], ADML [30], and SADL [27].

would become much more complex, with the attendant problems of ensuring that any observation derived from the architecture has a bearing on the system itself.

For this reason, two key elements of our approach are the explicit definition of style and its accessibility at run time for system adaptation. Specifically, we define a style as a system of types, plus a set of rules and constraints. The types are defined in Acme [12], a generic ADL that extends the above structural core framework with the notion of style. The rules and constraints are defined in Armani [26] a first-order predicate logic similar to UML's OCL [29], augmented with a small set of architectural functions. These functions make it easier to define logical expressions that refer to things like connectedness, type conformance, and hierarchical relationships.[3] We say that a system *conforms* to a style if it satisfies all of the constraints defined by the style (including type conformance).

An example of an architectural style is a pipe-filter style. Elements in this style include filter components, which receive data and transform that data, and pipe connectors, which transfer data between filters. In Acme, the definition of a filter component type looks like:

```
Component type Filter T = {
   Property throughput : float;
   Port stdIn : InputPortT;
   Port stdOut : OutputPortT;
}
```

This type definition would be instantiated in a given systems by creating specific filter components. Any component *conforming* to the FilterT type would have at least the throughput property, and the two ports stdIn and stdOut, which in turn need to conform to the port types InputPortT and OutputPortT.

Being able to define styles in Acme gives some reuse in our framework. We envision a suite of general styles (along with monitoring and repair capabilities) from which a style can be chosen to be plugged into our framework. An architect would then need to model the system according to this style, perhaps extending the style or utilizing other styles to model attributes of interest.[4]

4.2 Analytical Methods for Architectures

As we argued above, one of the main benefits of style-based architectural modeling is the ability to use analytical methods to evaluate properties of a system's architectural design. For example, MetaH uses real-time schedulability analysis, and Wright uses protocol model checking. Use of the appropriate analytical methods helps us to focus on the aspects of the architecture that we need to model, to identify the constraints of the style, and to guide the error resolution when constraints are violated. For instance,

---

[3] Details on Acme and Armani can be found elsewhere [12, 26]. Here we focus on how those representation schemes, originally developed as design-time notations, are extended and used to support run time adaptation.

[4] A style would also supply operators to modify the style, and perhaps repair facilities. These are discussed later in the section.

in a Service-Coalition style, cost analysis of the system indicates which services to monitor. Based on what factors drive cost—for example, performance—we can add to or refine cost-based constraints to take those factors into account. This can help guide us to the cause of error when a cost constraint fails. If performance were a factor, a cost violation in a particular component would suggest that we check the performance properties of that component for the cause. Furthermore, cost-benefit analysis would tell us how to trade-off cost with performance to find a better service during adaptation.

An analytical method can potentially be applied to several different styles. For example, one might use queuing theoretic analysis in a Client-Server style or a Pipe-Filter style, and cost-benefit analysis can be applied to almost any style. When applied to a particular style, however, the analytical method takes on the vocabulary of that style, and often augments elements of that style with analysis-specific properties. For example, queuing theoretic analysis augments a server component with properties such as load, service time, etc.

4.3 Using Styles to Assist Adaptation

The representation schemes for architectures and style outlined above were originally created to support design-time development tools. In this section we show how styles can be augmented to function as run time adaptation mechanisms. We then consider the supporting run time infrastructure needed to make this work out in practice (Section 4.4).

Two key augmentations to style definitions are needed to make them useful for run time adaptation: (1) the definition of a set of adaptation operators for the style, and (2) the definition of a set of repair strategies.

4.3.1 Adaptation Operators

The first extension is to augment a style description with a set of operators that define the ways one can change instances of systems in that style. Such operators determine a "virtual machine" that can be used at run time to adapt an architectural design.

Given a particular architectural style, there will typically be a set of natural operators for changing an architectural configuration and querying for additional information. In the most generic case, architectures can provide primitive operators for adding and removing components and connectors [31]. However, specific styles can often provide much higher-level operators that exploit the restrictions in that style and the intended implementation base. For example, a client-server style might support an operation to replicate a server to improve performance, whereas a pipe-filter style might support an operation to improve performance by adding a filer to compress the data on a pipe.

Two key factors determine the choice of operators for a style. First is the style itself – the kinds of components, connectors and configuration rules. Based on its constraints, a style can both limit the set of operations, and also suggest a set of higher-level operators. For example, if a style specifies that there must be exactly one instance of a particular type of component, such as a database, the style should not

provide operations to add or remove an existing instance of this type. On the other hand, if another constraint says that every client component in the system must be attached to the (unique) database, it would make sense that a "new-client" operation would automatically create a new client-database connector and attach it between the new component and the database. These style-specific operators are defined in terms of style-neutral operators such as "add a component" or "remove a connector." The definition of these style-neutral operations can be based on [40] or [41].

The second factor is the feasibility of carrying out the change. To evaluate feasibility requires some knowledge of the target implementation infrastructure. It makes no sense to prescribe an architectural operator that has no hope of ever being carried out on the running system. For some styles, the relation is defined by construction (since implementations are generated from architectures). More generally, however, the style designer may have to make certain assumptions about the availability of implementation-changing operators that will be provided by the run time environment of the system. (We return to this issue in Section 7.)

It is important to note that, while it is necessary to write adaptation operators for each style, we anticipate that this will only need to be done once for each style. A style should provide all operations that make sense in changing the style, regardless of any particular adaptation that might occur. For example, for a Client-Server style, the moveClient operator will be the same regardless of the adaptation being performed.

While adaptation operators are specific to styles we can, however, describe some, commonly occurring operators. In general, every style would be expected to have some form of add and remove, as well as possibly activate and deactivate operators for component instances (e.g., addClient, removeFilter, activateServer, deactivateDB). A style would also be expected to have add/remove or connect/disconnect operators to setup connectors between components (e.g., addRPC, removeVideoStream, connectPipe, disconnectSQL). In addition, there will typically be operators to create, delete, and modify element properties (e.g., createLatencyProperty, deleteFrameRateProperty, modifyCompressionProperty). Finally, depending on the style, there might conceivably be operators for changing a component's behavior via modification of specific properties of the component, such as changing the internal behavioral protocol of a component.

4.3.2 Repair Strategies

The second extension to the traditional notion of architectural style is the specification of repair strategies that correspond to selected constraints of the style. The key idea is that when a stylistic constraint violation is detected, the appropriate repair strategy will be triggered.

Describing Repair Strategies

A repair strategy has two main functions: first to determine the cause of the problem, and second to determine how to fix it. Thus the general form of a repair strategy is a sequence of repair *tactics*. Each repair tactic is guarded by a pre-condition that determines whether that tactic is applicable. The evaluation of a tactic's pre-condition will usually involve the examination of various properties of the architecture in order to pinpoint the problem and determine applicability. If it is applicable, the tactic exe-

cutes a repair script that is written as an imperative program using the style-specific operators described above.

To handle the situation that several tactics may be applicable, the enclosing repair strategy decides on the policy for executing repair tactics. It might apply the first tactic that succeeds. Alternatively, it might sequence through all of the tactics, or use some other style-specific policy.

The final complication associated with repair strategies is the use of transactions. The body of a repair strategy is typically enclosed within a transactional scope so that if an error occurs during the execution of a repair, the system can abort the repair, leaving the architecture in a consistent state. Failure of a repair strategy can be caused by a number of factors. For example, it may be the case that none of the tactics have applicable firing conditions. Or, an applicable tactic may find that conditions of the actual system or its environment do not permit it to carry out its repair script. Transaction aborts cause the system to inform the user of a system error that cannot be handled by the automated mechanisms.

Choosing Tactics

One of the principal advantages of allowing the system designer to pick an appropriate style is the ability to exploit style-specific analyses to determine whether repair tactics are sound. By sound, we mean that if executed, the changes will help reestablish the violated constraint.

In general, an analytical method for an architecture will provide a compositional method for calculating some system property in terms of the properties of its parts. For example, a reliability analysis will depend on the reliability of the architectural parts, while a performance analysis will depend on various performance attributes of the parts. By looking at the constraint to be satisfied, the analysis can often point the repair strategy writer both to the set of possible causes for constraint violation, and for each possible cause, to an appropriate repair.

For instance, one type of analysis appropriate to the pipe-filter style is throughput analysis. Such an analysis allows one to characterize a batch-processing pipe-filter system by the ratio of the input quantity to the output quantity (say, in terms of records), and compose the overall ratio from the ratio of each individual filter based on connection topology. The administrator of this system might want to enforce a constraint on the system in terms of this input-output ratio. Violation of this throughput ratio constraint suggests congestion of processing within the system. The associated repair strategy can then use a more fine-grained throughput analysis to pinpoint the segment or the particular filter causing the congestion.

4.4 Bridging the Gap to Implementation

As we have argued, the use of style allows us to provide automated support for architectural adaptation *at the model level.* That is, we can use the constraints, operators, and analytical methods to determine how to modify the architecture.

The only catch is that we somehow have to relate all of that to the real world. There are two parts to this. The first is getting information out of the executing system

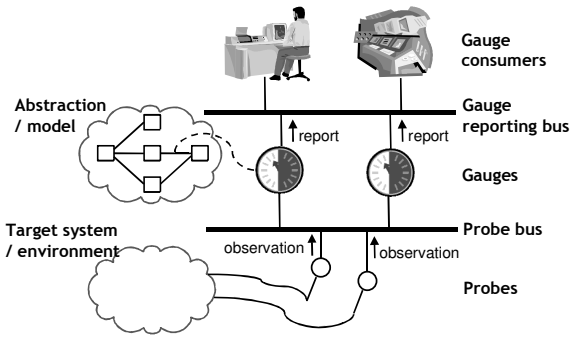

Fig. 3. Gauge Infrastructure.

so we can determine when architectural constraints are violated. The second is propagating architectural repairs into the system itself.

4.4.1 Monitoring

In order to provide a bridge from system level behavior to architecturally-relevant observations, we have defined a three-level approach illustrated in Figure 8. This monitoring infrastructure is described in more detail elsewhere [13]: here we summarize the main features, stressing the connection with style specifications.

The lowest level is a set of *probes*, which are "deployed" in the target system or physical environment.[5] Probes monitor the system and announce observations via a "probe bus." At the second level a set of *gauges* consumes and interprets 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 8 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.

The separation of the monitoring infrastructure into these parts helps isolate separable concerns. Probes are highly implementation-specific, and typically require detailed knowledge of the execution environment. Gauges are model-specific. They need only understand how to convert low-level observations into properties of more abstract representations, such as architectural models. Finally, gauge consumers are free to use the interpreted information to cause various actions to occur, such as displaying warnings to the user or automatically carrying out repairs.

In the context of architectural repair, we use the architectural style to inform us where to place gauges. Specifically, for each constraint that we wish to monitor, we must place gauges that dynamically update the properties over which the constraint is defined. In addition, our repair strategies may require additional monitored information to pinpoint sources of problems and execute repair operations.

---

[5] For monitoring, we utilize the terminology defined by the DASADA program, funded by DARPA.

While it may be necessary to develop gauges for each different style, and probes for each specific implementation, we can gain some leverage by using general monitoring technologies. For example, if the concerns are bandwidth or latency then it is possible to use general network gauges (for example, those based on Remos [13]) to report the bandwidth, regardless of the adaptation. Similarly, it is possible to use general probe technology to ameliorate the task of writing probes for particular implementations. For example, while it might be necessary to *choose* which particular method calls need to be monitored in a particular implementation, it is possible to use existing technologies like ProbeMeister [39] to generate the actual probes, without writing any additional code.

4.4.2 Repair Execution

The final component of our adaptation framework is a translator that interprets repair scripts as operations on the actual system (Figure 1, item 6). As we noted earlier, we assume that the executing system provides a set of system-changing operations via a Runtime Manager. The nature of these operations will depend heavily on the implementation platform. In general, a given architectural operation will be realized by some number of lower level system reconfiguration operations. Each such operator can raise exceptions to signal a failure. The Translator then propagates them to the model level, where transaction boundaries can cause the repair strategy to abort.

Even though the system-changing operations are system specific, the mechanisms for propagating system changes can be fairly general, subject to the constraints of the implementation platform. These mechanisms can be as simple as socket communication, RPC, or Java RMI, or as complicated as mobile-code or an entire change propagation technology.

4.4.3 Putting the Pieces Together

Let us summarize how the parts work together, end-to-end, and how pieces of the framework in Figure 1 interact. While the system is running, relevant system properties are observed and collected by gauges in the Monitoring Mechanisms and updated on the Architectural Model. Whenever there is a change in a gauge value, the Analyzer in the Architecture Manager re-evaluates the architectural constraints to check for violation. Suppose that a latency constraint violation is detected in some Client role, then the Analyzer calls the Repair Handler to trigger a repair. The Repair Handler first signals the Analyzer to suspend all monitoring and captures a "snapshot" of the current state of the Architectural Model – doing so prevents other constraint violation from interfering with the present repair and preserves the property values at the time of constraint violation to facilitate decision-making. The Repair Handler then begins running the repair script.

The Repair Handler executes repair scripts, which involve calls to the style operators. These calls are executed by the Style API Interpreter, which interprets the calls as primitive architectural operators to update the Architectural Model (via the Generic API). The Style API Interpreter also passes the style operator calls to the Translator.

The Translator translates architectural style operations into implementation operations and passes them to the Runtime Manager, which executes it to make changes to the Executing System. The implementation operations have exceptions not shown that may be raised if execution fails. The Translator would then pass the exception signal

back to the Repair Handler, which aborts the repair transaction. Whether the repair transaction commits or aborts the Repair Handler signals to the Analyzer to resume system monitoring and resets appropriate gauges.

At this point, as part of the dynamic verification to ensure that the repair was effective, the constraints are re-evaluated to determine whether any violations are now fixed, and the repair cycle completes. If a violation remains, or if a new violation is detected, the repair is triggered again and the process repeats.

5. Performance Adaptation of a Web-Based Server-Client System

In this section we give a detailed end-to-end description of how each of the elements in our adaptation framework come together to achieve runtime adaptation. We use the example described in Section 3 to illustrate our technique. The example is simple load balancing of a web-based client-server system. This is example is used simply to illustrate how our technique works; we are not proposing that this technique be applied to load-balancing of such systems – a technique that is already embedded in many systems.

```
Family PerformanceClientServerFam extends ClientServerFam with {
   Component Type PAClientT extends ClientT with {
      Properties  {
         Requests : sequence <any>;
         ResponseTime : float;
         ServiceTime : float;
      };
   };
   Connector Type PALinkT extends LinkT with {
      Properties {
         DelayTime : float;
      };
   };
   Component Type PAServerGroupT extends ServerGroupT with {
      Properties {
         Replication : int <<default : int = 1;>>;
         Requests : sequence <any>;
         ResponseTime : float;
         ServiceTime : float;
         AvgLoad : float;
      };
      Invariant AvgLoad > minLoad;
   };
   Role Type PAClientRoleT extends ClientRoleT with {
      Property averageLatency : float;
      Invariant averageLatency < maxLatency;
   };

   Property maxLatency : float;
   Property minLoad : float;
};
```

Fig. 4. Client/Server Style Extended for Analysis.

```
Family ClientServerFam = {
   Component Type ClientT = {…};
   Component Type ServerT = {…};

   Component Type ServerGroupT = {…};

   Role Type ClientRoleT = {…};
   Role Type ServerRoleT = {…};

   Connector Type LinkT = {
      invariant size(select r : role in Self.Roles |
            declaresType(r, ServerRoleT)) == 1;
      invariant size(select r : role in Self.Roles |
            declaresType(r, ClientRoleT)) >= 1;
      Role ClientRole1 : ClientRoleT;
      Role ServerRole : ServerRoleT;
   };
};
```

Fig. 5. Client/Server Style Definition.

5.1 Defining a Client-Server Architectural Style

Figure 4 contains a partial description of the style used to characterize the class of web-based systems of our example. The style is actually defined in two steps. The first step specifies a generic client-server style (called a *family* in Acme). It defines a set of component types: a web client type (ClientT), a server group type (ServerGroupT), and a server type (ServerT). It also defines a connector type (LinkT). Constraints on the style (appearing in the definition of LinkT) guarantee that the link has only one role for the server. Other constraints, not shown, further define structural rules (for example, that each client must be connected to a server).

There are potentially many possible kinds of analysis that one might carry out on client-server systems built in this style. Since we are particularly concerned with overall system performance, we augment the client-server style to include performance-oriented properties. These include the response time and degree of replication for servers and the delay time over links. This style extension is shown in Figure 5. Constraints on this style capture the desired performance related behavior of the system. The first constraint, associated with PAServerGroupT, specifies that a server group should not be under-utilized. The second constraint, as part of the PAClientRoleT, specifies that the latency on this role should not be above some specified maximum.

Having defined an appropriate style, we can now define a particular system configuration in that style, such as the one illustrated in Figure 6.

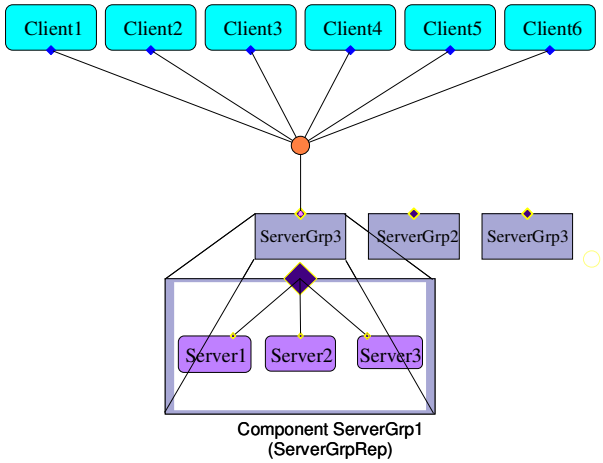

Fig. 6. Architectural Model of Example System.

5.2 Using M/M/m Performance Analysis to Set Initial Conditions

The use of buffered request queues, together with replicated servers, suggests using queuing theory to understand the performance characteristics of systems built in the client-server style above. As we have shown elsewhere [35], for certain architectural styles queuing theory is useful for determining various architectural properties including system response time, server response time (T_s), average length of request queues (Q_s), expected degree of server utilization (u_s), and location of bottlenecks.

In the case of our example style, we have an ideal candidate for M/M/m analysis. The *M/M* indicates that the probability of a request arriving at component *s,* and the probability of component *s* finishing a request it is currently servicing, are assumed to be exponential distributions (also called "memoryless," independent of past events); requests are further assumed to be, at any point in time, either waiting in one component's queue, receiving service from one component, or traveling on one connector. The *m* indicates the replication of component *s*; that is, component *s* is not limited to representing a single server, but rather can represent a server group of *m* servers that are fed from a single queue. Given estimates for clients' request generation rates and servers' service times (the time that it takes to service one request), we can derive performance estimates for components according to Table 1. To calculate the expected system response time for a request, we must also estimate the average delay D_c imposed by each connector *c*, and calculate, for each component *s* and connector *c*, the average number of times (V_s, V_c) it is visited by that request. (Given V_s and the rates at which client components generate requests, we can derive rather than estimate R_s, the rate at which requests arrive at server group *s*.)

Table 1. Performance Equations from [4]

(1)	Utilization of server group *s*	$u_s = \frac{R_s S_s}{m}$
(2)	Probability {no servers busy}	$p_0 = \left[\sum_{i=0}^{m} \frac{(mu_s)^i}{i!} + \frac{u_s (mu_s)^m}{m!(1-u_s)} \right]^{-1}$
(3)	Probability {all servers busy}	$P_Q = \frac{p_0 (mu_s)^m}{m!(1-u_s)}$
(4)	Average queue length of *s*	$Q_s = \frac{P_Q u_s}{1-u_s}$
(5)	Average response time of *s*	$T_s = S_s + \frac{P_Q u_s}{R_s(1-u_s)} =$ $S_s + \frac{S_s (mu_s)^m}{mm!(1-u_s)^2 \sum_{n=0}^{m} \frac{(mu_s)^n}{n!} + (1-u_s)(mu_s)^{m+1}}$
(6)	System response time (latency)	$\sum T_s V_s + \sum D_c V_c$

Applying this M/M/m theory to our style tells us that with respect to the average latency for servicing client requests, the key design parameters in our style are (a) the replication factor *m* of servers within a server group, (b) the communication delay *D* between clients and servers, (c) the arrival rate *R* of client requests and (d) the service time *S* of servers within a server group.

In previous work [35] we showed how to use this analysis to provide an initial configuration of the system based on estimates of these four parameters. In particular, Equation (5) in Table 1 indicates for each server group a design tradeoff between utilization (underutilized servers may waste resources, but provide faster service) and response time. Utilization is in turn affected by service time and replication. Thus, given a range of acceptable utilization and response time, if we choose service time then replication is constrained to some range (or vice versa). As we will show in the next section, we can also use this observation to determine sound run time adaptation policies.

We can use the performance analysis to decide the following questions about our architecture, assuming that the requirements for the initial system configuration are that for six clients each client must receive a latency not exceeding 2 seconds for each request and a server group must have a utilization of between 70% and 80%:

- How many replicated servers must exist in a server group so that the server group is properly utilized?
- Where should the server group be placed so that the bandwidth (modeled as the delay in a connector) leads to latency not exceeding 2 seconds?

Given a particular service time and arrival rate, performance analysis of this model gives a range of possible values for server utilization, replication, latencies, and system response time. We can use Equation (5) to give us an initial replication count and Equation (6) to give us a lower bound on the bandwidth. If we assume that the arrival rate is 180 requests/sec, the server response time is between 10ms and 20ms the average request size is 0.5KB, and the average response size is 20KB, then the performance analysis gives us the following bounds:

Initial server replication count= 3-5
Zero-delay System Response Time = 0.013-0.026 seconds

Therefore,

0 < Round-trip connector delay < 1.972 seconds, or
0 < Average connector delay < .986 seconds

Thus, the average bandwidth over the connector must be greater than 10.4KB/sec. This analysis provides several key criteria for monitoring the running system. First, if latency increases undesirably, then we should check to ensure that the bandwidth assumption still holds between a client and its server. Second, if bandwidth is not the causing factor, then we should examine the load on the server.

5.3 Defining Adaptation Operators

The client-server architectural style suggests a set of style-specific adaptation operators that change the architectural while ensuring the style constraints. These operators are:

- **addServer**(): This operation is applied to a component of type ServerGroupT and adds a new component of type ServerT to its representation, ensuring that there is a binding between its port and the ServerGroup's port.
- **move**(to:ServerGroupT): This operation is applied to a client and first deletes the role currently connecting the client to the connector that connects it to a server group. It then performs the necessary attachment to a LinkT connector that will connect it to the server group passed in as a parameter. If no such connector exists, it will create one and connect it to the server group.
- **remove**(): This operation is applied to a server and deletes the server from its containing server group. Furthermore, it changes the replication count on the server group and deletes the binding.

The above operations all effect changes to the model. The next operation queries the state of the running system:

- **findGoodSGroup**(cl:ClientT,bw:float):ServerGroupT; finds the server group with the best bandwidth (above *bw*) to the client *cli*, and returns a reference to the server group.

These operators reflect the considerations just outlined. First, from the nature of a server group, we get the operations of adding or removing a server from a group. Also, from the nature of the asynchronous request connectors, we get the operations

of adapting the communication path between particular clients and server groups. Second, based on the knowledge of supported system change operations, outlined in Section 4.4, we have some confidence that the architectural operations are actually achievable in the executing system.

5.4 Defining Repair Strategies to Maintain Performance

Recall that the queuing theory analysis points to several possible causes for why latency could increase. Given these possibilities, we can show how the repair strategy developed from this theoretical analysis. The equations for calculating latency for a service request (Table 1) indicate that there are four contributing factors: (1) the connector delay, (2) the server replication count, (3) the average client request rate, and (4) the average server service time. Of these we have control over the first two. When the latency is high, we can decrease the connector delay (by moving clients to servers that are closer) or increase the server replication count to decrease the latency. Determining which tactic depends on whether the connector has a low bandwidth (inversely proportional to connector delay) or if the server group is heavily loaded (inversely proportional to replication). These two system properties form the preconditions to the tactics; we have thus developed a repair strategy with two tactics.

Applying the Approach

We specify repair strategies using a repair language that supports basic flow control, Armani constraints, and simple transaction semantics. Each constraint in an architectural model can be associated with a repair strategy, which in turn employs one or more repair tactics.

Figure 7 (lines 1-3) illustrates the repair strategy associated with the latency threshold constraint. In line 2, "!→" denotes "if constraint violated, then execute." The top-level repair strategy in lines 5-17, fixLatency, consists of two tactics. The first tactic in lines 19-31 handles the situation in which a server group is overloaded, identified by the precondition in lines 24-26. Its main action in lines 27-29 is to create a new server in any of the overloaded server groups. The second tactic in lines 33-48 handles the situation in which high latency is due to communication delay, identified by the precondition in lines 34-36. It queries the running system to find a server group that will yield a higher bandwidth connection in lines 40-41. In lines 42-44, if such a group exists it moves the client-server connector to use the new group. The result of an instance of this repair on Figure 6 is depicted in Figure 8. The repair strategy uses a policy in which it executes these two tactics sequentially: if the first tactic succeeds it commits the repair strategy; otherwise it executes the second. The strategy will abort if neither tactic succeeds, or if the second tactic finds that it cannot proceed since there are no suitable server groups to move the connection to.

```
01 invariant r.averageLatency <= maxLatency
02 !→
03     fixLatency(r);
04
05 strategy fixLatency (badRole: ClientRoleT) = {
06     begin repair-transaction;
07     let badClient: ClienT =
08         select one cli: ClientT in self.Components |
09             exists p: RequestT in cli.Ports | attached(badRole, p);
10     if (fixServerLoad(badClient)) {
11         commit repair-transaction;
12     else if (fixBandwidth(badClient, badRole) {
13         commit repair-transaction;
14     } else {
15         abort(ModelError);
16     }
17 }
18
19 tactic fixServerLoad (client: ClientT) : boolean = {
20     let overloadedServerGroups: Set{ServerGroupT} =
21         { select sgrp: ServerGroupT in self.Components |
22             connected(sgrp, client) and
23             sgrp.AvgLoad > maxServerLoad };
24     if (size(overloadedServerGroups) == 0) {
25         return false;
26     }
27     foreach sGrp in overloadedServerGroups {
28         sGrp.addServer();
29     }
30     return (size(overloadedServerGroups) > 0);
31 }
32
33 tactic fixBandwidth (client: ClientT, role: ClientRoleT) : boolean = {
34     if (role.Bandwidth >= minBandwidth) {
35         return false;
36     }
37     let oldSGrp: ServerGroupT =
38         select one sGrp: ServerGroupT in self.Components |
39             connected(client, sGrp);
40     let goodSGrp: ServerGroupT =
41         findGoodSGrp(client, minBandwidth);
42     if (goodSGrp != nil) {
43         client.moveClient(oldSGrp, goodSGrp);
44         return true;
45     } else {
46         abort(NoServerGroupFound);
47     }
48 }
```

Fig. 7. Repair Tactic for High Latency.

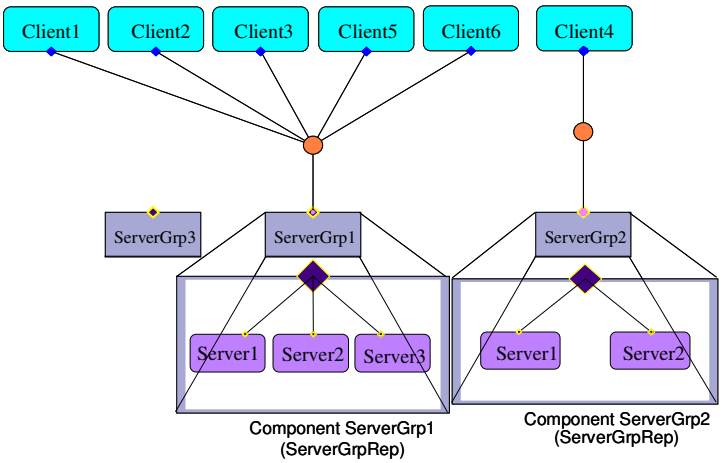

Fig. 8. Model of System After Low Bandwidth Repair.

5.5 Style-Based Monitoring

In our example above we are concerned with the average latency of client requests. To monitor this property, we must associate a gauge with the averageLatency property of each client role (see the definition of ClientRoleT in Figure 4). This latency gauge in turn deploys a probe into the implementation that monitors the timing of reply-request pairs. When it receives such monitored values it averages them over some window, updating the latency property in the architecture model when it changes. The latency gauge that we use is not specific to this style, or indeed to this implementation. The gauges utilizes probes that use the Remos network monitoring service, which in turn uses the SNMP to ascertain properties of the network.

But average latency is not the only architectural property that we need to monitor. The repair tactics, derived from queuing theoretic model of performance analysis, rely on information about two additional constraints: whether the bandwidth between the client and the server is low or whether the server group is overloaded (or both). Thus, to determine why latency is high in the architecture, we need to monitor these two properties. The gauge for measuring bandwidth uses the same probe used by the latency gauge for measuring the time it takes to receive a reply. An additional probe measures the size of the reply and calculates the bandwidth based on these values. Determining the load on the server can be done in a number of ways. We measure the size of a request queue to indicate whether the server group is overloaded.

5.6 Mapping Architectural Operators to Implementation Operators

To illustrate, the specific operators and queries supported by the Runtime Manager in our example are listed in Table 2. These operators include low-level routines for creating new request queues, activating and deactivating servers, and moving client communications to a new queue.

The Translator for our example maps the Style API Interpreter operations described in Section 4.3.1 to the Runtime Manager operations using the scheme summarized in Table 2. (Parameters passed between the levels also need to be translated. We do not discuss this here.) The actual map involves mapping model-level parameters to implementation level parameters, and mapping return values to model values.

5.7 Putting the Pieces Together

As an example of how the adaptation framework fits together in our implementation, we will consider one cycle of the repair, starting with a latency probe reporting a value, and ending with a client moving to a new server group. This cycle indicates how the architecture in Figure 6 is transformed into the architecture in Figure 8.

1. The bandwidth probe on the link between Client4 and ServerGroup1 reports a bandwidth of 18KB/sec to the probe bus.
2. The latency gauge attached to Client4's role combines this value with the average size of requests that it has seen, and calculates an average latency of 2.5secs, which it reports to the gauge bus. Similarly, the bandwidth gauge attached to Client4's role reports a bandwidth of 18KB/sec to the gauge bus.
3. The Architecture Manager, implemented as a gauge consumer, receives these values and adjusts the averageLatency and bandwidth properties of Client4's role.
4. The Analyzer, implemented using our Armani constraint analyzer, reevaluates constraints. The constraint averageLatency < maxLatency in Client4's role fails.
5. Tailor, the repair handler, is invoked and pauses monitoring before starting to execute the repair strategy in Figure 7, passing Client4's role as a parameter.
6. The repair strategy first attempts to fix the server load, but returns false because no

Table 2. Mapping Between Architecture and Implementation Operations

Model Level	Environment Level
addServer	**findServer** **activateServer** **connectServer**
moveClient	**createReqQue** **moveClient**
findGoodSGrp	Conditionals + multiple calls to **remos_get_flow**

servers are overloaded.

7. The repair strategy attempts to fix the bandwidth. It examines the bandwidth property of the role, and determines that it is larger than 10.4KB/sec (line 34). It then calls the architectural operator findGoodSGrp to find the server group with the best bandwidth. This invokes queries to remos_get_flow.
8. The operator findGoodSGrp returns ServerGroup2 now has the best bandwidth and initiates the moveClient operator (line 43). This in turn invokes the change interface for the application to effect the move.

6. Implementation Status

In terms of the adaptation framework in Figure 1, our implementation contains the following pieces:

Monitoring Mechanisms: Our approach is general enough to be used with existing technologies for monitoring systems and their environments. To connect with the infrastructure described in Section 4.4.1, a wrapper needs to be written for these technologies that allows events to be generated according to the probe infrastructure, mentioned in Figure 8, turning the technology into a probe. We have developed prototype probes for gathering information out of networks, based on the Remos system [21]. We have developed general-purpose gauges that can be used to report data about expected and observed bandwidth and latencies based on data from this system.

Other technology has also been successfully integrated into our infrastructure, most notably the ProbeMeister system for unobtrusively monitoring Java classes [39], and the Event Packager and Event Distiller systems for monitoring temporal events from executing systems [17]. In addition, we have produced gauges that monitor the adherence of elements of the architecture to protocols expressed in FSP [23].

Architectural Models: AcmeStudio, a design environment that allows architectures to be described in Acme, has been modified so that it provides run time observation of a software architecture [34]. A general library has been developed that can be integrated with other architectural tools to associate gauge information with architectural models.

Architectural Analysis: We have modified our tool for evaluating Armani constraints at design times so that it evaluates constraints dynamically at run time.

Repair Handler: The Armani constraint evaluator has been augmented so that it supports the specification and execution of repairs.

Translator and **Runtime Manager:** Currently, we have hand-tailored support for these components that need to be changed for each implementation. Our work in this area will concentrate on providing more general mechanisms where appropriate, and perhaps using off-the-shelf reconfiguration commands for commercial systems. In fact, we are actively investigating how to utilize the Workflakes system for a more general solution to the problem of mapping between architecture and implementation.

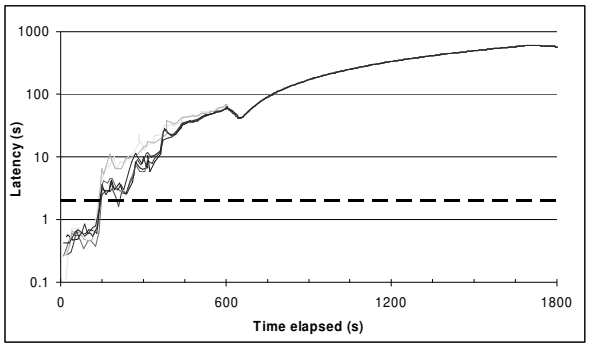

Fig. 9. Average Latency for Control (No Repair).

7. Experience

Thus far we have experimented with architectural adaptation for two kinds of system properties: (1) performance for web-based systems, illustrated earlier, and (2) protocol conformance.

To evaluate the effectiveness of our adaptation framework for performance-oriented adaptation, we conducted an experiment to test system adaptation using a dedicated, experimental testbed consisting of five routers and eleven machines communicating over 10 Mbps lines. The implementation that we used for our experiment was based on the example presented in this paper – that of a client-server system using replicated server groups communicating over a distributed system. System loads were fabricated in three segments over 30 minutes so that we could observe the self-repair behavior of the system.

The results showed that for this application and the specific loads used in the experiment, self-repair significantly improved system performance. Figures 9 and 10 show sample results for the system performance without adaptation, and with, respectively. (See [7] for details.) However, it also revealed, perhaps not unexpectedly, that externalized repair introduces some significant latency. In our system it took several seconds for the system to notice a performance problem and several more seconds to fix it. Although we can imagine speeding up the roundtrip repair time, this does indicate that the approach is best suited for repair that operates on a global scale, and that handles longer term trends in system behavior.

The second application of the approach has been to monitor and check protocols of interaction between components. Connectors are associated with protocol constraints that indicate the allowed order of communication events. These are defined in a process algebra, FSP [23], and then used by "protocol gauges" at run time to detect when communicating components fail to respect the specified protocols. For example, a protocol error might occur when a component attempts to write data to a pipe after it has closed that pipe, or if a client attempts to communicate with a server without first initializing its session.

8. Discussion

We have described an approach in which architecture-based self-adaptation is supported by the incorporation of styles as an explicit design choice in the adaptation framework. The flexibility inherent in this approach permits the system maintainer to pick a style that matches well to existing implementation bases, provides a formal basis for specifying constraints, and can permit the definition of repair policies that are justified by analytic methods.

However, this flexibility also introduces several new complexities over other approaches in which the choice of architectural style is hardwired into the framework. In particular, at least three critical questions are raised: First, is it always possible to map architectural repairs into corresponding system changes? Second, is it always possible to monitor relevant run time information? Third, is it reasonable to expect that analytical techniques can address a sufficiently broad set of concerns to inform our repair strategies? We address each issue in turn.

Model-Implementation Map: In our approach the ability to map architectural changes to corresponding implementation reconfigurations is moderated by two factors. First is an assumption that systems provide a well-defined set of operations for modifying a running system. Of course, in general this may not be true. Some systems

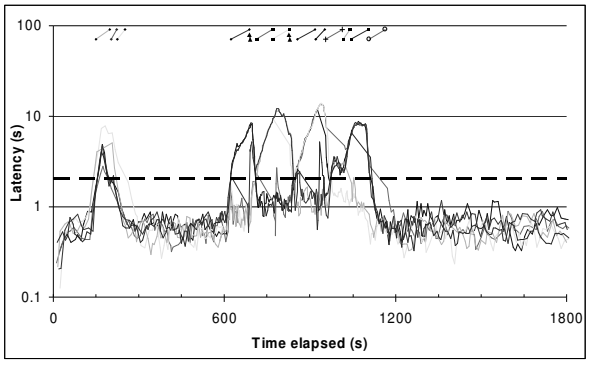

Fig. 10. Average Latency under Repair.

are inherently not reconfigurable, in which case our approach would simply not work. However, many systems do in fact embody changing operations – such as the ability to load dynamic libraries and remote code, to redirect communications to alternative pathways, or to do dynamic resource management. Moreover, we would argue that such capabilities are going to be increasingly prevalent in modern systems that are intended to function in a connected, service-based universe. For example, modern frameworks like Jini provide as a fundamental building block the notion of allocation and deallocation of resources, and location-independence of services.

The other moderating factor is an assumption that architectural style is not chosen arbitrarily. Obviously, attempting to pair an arbitrary style with an arbitrary implementation could lead to considerable difficulty in relating the two. However, one of the hallmarks of our approach is that it encourages one to match an appropriate style to an implementation base. Hence, in fact, the flexibility of choosing a style can actually help reduce the gap between system implementations and architectural models.

Implementation-Model Map: For our approach to work it must be possible to reflect dynamic system state into an architectural model. To do this we provide a multi-leveled framework that separates concerns of low-level system instrumentation from concerns of abstracting those results in architecturally meaningful terms. What makes us think that either part will be feasible in the general case?

The ability to monitor systems is itself an active research area. Increasingly systems are expected to provide information that can be used to determine their health. Moreover, there is an increasingly large number of non-intrusive post-deployment monitoring schemes. For example, to deal with network performance we were able to use a monitoring infrastructure developed completely independently. It in turn relies on the standard protocol SNMP. Other researchers and practitioners are developing many other schemes such as the ability to place monitors between COM components, the ability to monitor network traffic to determine security breaches, the ability to monitor object method calls, and various probes that determine whether a given component is alive.

In terms of mapping low-level information to architectural information, the capability will certainly depend on the distance between the architectural and implementation styles. As we argued earlier, our approach encourages developers to pick styles where that mapping will be straightforward.

Analytical Methods: A key feature of our approach is the notion that repair strategies should leverage architectural analyses. We demonstrated one such analysis for performance. What makes us think that others exist? In fact, there is considerable work recently on finding good architecture-based analyses. For example, Klein et al. [19] provide a method of reasoning about the behavior of component types that interact in a defined pattern. In earlier work we showed how to adapt protocol analysis to architectural modification [3]. Others have shown how real-time schedulability can be applied [38]. Although far from providing a complete repertoire of analytical techniques, the space is rich, and getting richer.

9. Conclusion and Future Work

In this paper we have presented a technique for using software architectural styles to automate dynamic repair of systems. In particular, styles and their associated analyses

- make explicit the constraints that must be maintained in the face of evolution
- direct us to the set of properties that must be monitored to achieve system quality attributes and maintain constraints
- define a set of abstract architectural operators for repairing a system
- allow us to select appropriate repair strategies, based on analytical methods

We illustrated how the technique can be applied to performance-oriented adaptation of certain web-based systems.

For future research we see opportunities to improve each of the areas mentioned in Section 7. We need to be able to develop mechanisms that provide richer adaptability for executing systems. We need new monitoring capabilities, and reusable infrastructure for relating monitored values to architectures. We need new analytical methods for architecture that will permit the specification of principled adaptation policies.

Additionally we see a number of other key future research areas. First is the investigation of more intelligent repair policy mechanisms. For example, one might like a system to dynamically adjust its repair tactic selection policy so that it takes into consideration the history of tactic effectiveness: effective tactics would be favored over those that sometimes fail to produce system improvements. Second is the link between architectures and requirements. Systems may need to adapt, not just because the underlying computation base changes, but because user needs change. This will require ways to link user expectations to architectural parameters and constraints. Third is the development of concrete instances of our approach for some of the common architectural frameworks, such as EJB, Jini, and CORBA.

Acknowledgements

The research described in this paper was supported by DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616. 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.

References

[1] Agha, G. A. Adaptive Middleware. Communications of the ACM 45(6):30-32, Jun. 2002.

[2] Allen, R.J. A Formal Approach to Software Architecture. PhD Thesis, published as Carnegie Mellon University School of Computer Science Technical Report CMU-CS-97-144, May 1997.

[3] Allen, R.J., Douence, R., and Garlan, D. Specifying Dynamism in Software Architectures. Proc. the Workshop on Foundations of Component-Based Software Engineering, Sept. 1997.

[4] Allen, R.J and Garlan, D. A Formal Basis for Architectural Connection. ACM Transactions of Software Engineering and Methodology, Jul. 1997.

[5] Bertsekas, D. and Gallager, R. Data Networks, Second Edition. Prentice Hall, 1992. ISBN 0-13-200916-1.

[6] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. Proc. the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, Jul. 2000.

[7] Cheng, S-W., Garlan D., Schmerl, B.R., Steenkiste, P.R., Hu. N. Software Architecture-based Adaptation for Grid Computing. Proc. the 11th IEEE Conference on High Performance Distributed Computing (HPDC'02), Edinburgh, Scotland, Jul. 2002.

[8] Dashofy, E., Garlan, D., van der Hoek, A., and Schmerl, B. http://www.ics.uci.edu/pub/arch/xarch/.

[9] Dashofy, E., van der Hoek, A., and Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. Proc. the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, Aug. 2001.

[10] Gantenbien, R.E. Dynamic Binding in Strongly Typed Programming Languages. *Journal of Systems and Software* **14**(1):31-38, 1991.

[11] Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. Proc. the SIGSOFT '94 Symposium on the Foundations of Software Engineerng, , New Orleans, LA, Dec. 1994.

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

[13] Garlan, D., Schmerl, B.R., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. Proc. the 1st Working Conference on Complex and Dynamic System Architecture. Brisbane, Australia, Dec. 2001.

[14] Gorlick, M.M., and Razouk, R.R. Using Weaves for Software Construction and Analysis. Proc. the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991.

[15] Gorlick, M.M. Distributed Debugging on $5 a day. Proc. the California Software Symposium, University of California, Irvine, CA, 1997 pp. 31-39.

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

[17] Gross, P.N, Gupta, S., Kaiser, G.E., Kc, G.S., and Parekh, J.J. An Active Events Model for Systems Monitoring. Proc. the 1st Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, Dec. 2001.

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

[19] Klein, M., Kazman, R., Bass, L., Carriere, J., Barbacci, M., Lipson, H. Attribute-Based Architecture Styles. Software Architecture Proc. the First Working IFIP Conference on Software Architecture (WICSA1), (San Antonio, TX), Feb. 1999, 225-243.

[20] Kon, F., Romn, M., Liu, P., Mao, J., Yamane, T., Magalh, C.,Campbell, R.H. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. IFIP/ACM International Conference on Distributed Systems Platforms, 2000, New York, New York.

[21] Lowekamp, B., Miller, N., Sutherland, D., Gross, T., Steenkiste, P., and Subhlok, J. A Resource Query Interface for Networ-aware Applications. Cluster Computing, **2**:139-151, Baltzer, 1999.

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

[23] Magee, J., and Kramer, J. Concurrency: State Models and Java Programs. Wiley, 1999.

[24] Métayer, D.L. Describing Software Archtiecture Styles using Graph Grammars. *IEEE Transactions on Software Engineering*, **24**(7):521-553, Jul. 1998.

[25] Miller, N., and Steenkiste, P. Collecting Network Status Information for Network-Aware Applications. IEEE INFOCOM 2000, Tel Aviv, Israel, Mar. 2000.

[26] Monroe, R.T. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.

[27] Moriconi, M. and Reimenschneider, R.A. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI International, Mar. 1997.

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

[29] Object Management Group. The OMG Unified Modeling Language Specification, Version 1.4. Sep. 2001. Available at http://www.omg.org/technology/documents/formal/uml.htm.

[30] The OpenGroup. Architecture Description Markup Language (ADML) Version 1. Apr. 2000. Available at http://www.opengroup.org/publications/catalog/i901.htm.

[31] Oriezy, P., Medvidovic, N., and Taylor, R.N. Architecture-Based Runtime Software Evolution. Proc. the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, Apr. 1998, pp. 11—15.

[32] Oriezy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An Architecture-Based Approach to Self-Adaptive Software. *IEEE* Intelligent *Systems* **14**(3):54-62, May/Jun. 1999.

[33] Shaw, M. and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[34] Schmerl, B.R., and Garlan, D. Exploiting Architectural Design Knowledge to Support Self-repairing Systems. Proc, the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, Jul. 15-19, 2002.

[35] Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proc. the 1998 Conference on Software Engineering and Knowledge Engineering, Jun. 1998.

[36] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* **22**(6):390-406, 1996.

[37] Valetto, G., and Kaiser, G. A Case Study in Software Adaptation. Proc. the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), Charleston, SC, Nov. 2002.

[38] Vestel, S. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, Apr. 1996.

[39] Wells, D., and Pazandak, P. Taming Cyber Incognito: Surveying Dynamic / Reconfigurable Software Landscapes. Proc. the 1st Working Conference on Complex and Dynamic Systems Architectures,, Brisbane, Australia, Dec 12-14, 2001.

[40] Wermelinger, M., Lopes, A., and Fiadeiro, J.L. A Graph Based Architectural (Re)configuration Language. Proc. the Joint 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Vienna, Austria, Sep. 2001, pp. 21—32.

[41] Wile, D.S. AML: An Architecture Meta-Language. Proc. the Automated Software Engineering Conference, Cocoa Beach, FL, Oct. 1999.