AFRL-IF-RS-TR-2005-121
Final Technical Report
April 2005

RAINBOW: ARCHITECTURE-BASED ADAPTATION OF COMPLEX SYSTEMS

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K501

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-121 has been reviewed and is approved for publication

APPROVED: /s/

DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2005	Final Jun 00 – Oct 03

4. TITLE AND SUBTITLE
RAINBOW: ARCHITECTURE-BASED ADAPTATION OF COMPLEX SYSTEMS

5. FUNDING NUMBERS
C - F30602-00-2-0616
PE - 62302E
PR - DASA
TA - 00
WU - 01

6. AUTHOR(S)
David Garlan and
Bradley Schmerl

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh Pennsylvania 15213-3890

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFT
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-121

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Deborah A. Cerino/IFT/(315) 330-1445/ Deborah.Cerino@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
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. Traditionally system self repair has been handled within the application, and at the code level. An alternative approach, and the approach taken under this effort, is to use architectural models, maintained at run time, as the basis for system reconfiguration and repair. 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 the problem. Moreover, architectural models can make integrity constraints explicit, helping to ensure the validity of any system change. This effort demonstrated how to generalize architecture-based adaptation by making the choice of architectural style an explicit design parameter in the framework. This allows system designers to pick an appropriate architectural style to expose properties of interest, provide analytic leverage and map cleanly to existing implementations and middleware.

14. SUBJECT TERMS
Architecture-Based Adaptation, System Self-Repair

15. NUMBER OF PAGES
49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents

1. Introduction 1
1.1. Innovative Claims 2
2. Approach and Framework 4
2.1. Architectures and Architectural Style 7
2.1.1. Analytical Methods for Architectures 9
2.2. Monitoring 9
2.3. Analysis 10
2.4. Reconciliation 10
2.4.1. Adaptation Operators 11
2.4.2. Repair Strategies 12
2.5. Propagation 13
3. Tool support 13
3.1. Gauge Infrastructure 13
3.1.1. Gauge Definition 14
3.1.2. Implementation 17
3.1.3. Gauge Workbench 20
3.2. Gauges 20
3.2.1. Network Performance Gauges 20
3.2.2. Protocol Gauges 21
3.2.3. Architecture Gauges 21
3.3. Repair 21
3.4. Integration with Acme Tools 21
3.4.1. Changes to Existing Tools 22
3.4.2. New Architecture Tools 23
3.4.3. Integrating Architectural Tools 24
4. Case Studies 26
4.1. Performance-based Adaptation of a Web-based Client-Server System 26
4.1.1. Defining a Client-Server Architectural Style 26
4.1.2. Using M/M/m Performance Analysis to Set Initial Conditions 27
4.1.3. Defining Adaptation Operators 30

4.1.4. Defining Repair Strategies to Maintain Performance 30
4.1.5. Style-Based Monitoring 32
4.1.6. Mapping Architectural Operators to Implementation Operators 33
4.1.7. Putting the Pieces Together 33
4.1.8. Results 34
4.2. Performance Adaptation of GeoWorlds 39
5. Conclusions and Future Work 39
6. Publications 40
References 41

List of Figures.

Figure 1. Adaptation Framework. 5
Figure 2. Deployment Architecture of Example System. 5
Figure 3. Gauge Infrastructure. 9
Figure 4. Gauge Example. 14
Figure 5. Attaching Gauges to Systems in Acme. 18
Figure 6. Gauge Infrastructure Implementation. 19
Figure 7. Generating Gauges from Acme Descriptions. 20
Figure 8. Integration of Architecture Tools. 25
Figure 9. Client/Server Style Definition. 26
Figure 10. Client/Server Style Extended for Analysis. 27
Figure 11. Architectural Model of Example System. 28
Figure 12. Repair Tactic for High Latency. 31
Figure 13. Model of System After Low Bandwidth Repair. 34
Figure 14. The Experimental Testbed. 35
Figure 15. Bandwidth and Server Load Generation. 36
Figure 16. Average Latency for Control. 37
Figure 17. Server Load for Control. 37
Figure 18. Available Bandwidth in Control. 37
Figure 19. Average Latency under Repair. 37
Figure 20. Server Load under Repair. 37
Figure 21. Available Bandwidth with Repair. 37

List of Tables

TABLE 1. AN EXAMPLE OF GAUGE TYPE SPECIFICATION. 16
TABLE 2. AN EXAMPLE OF GAUGE INSTANCE SPECIFICATION. 17
TABLE 3. PERFORMANCE EQUATIONS FROM [4]. 29
TABLE 4. MAPPING BETWEEN ARCHITECTURE AND IMPLEMENTATION OPERATIONS. 33

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 Remote Procedure Calls). 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 [17]. The Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) Project, funded by DARPA, seeks to mature this approach to enable mission-critical systems to meet the high assurance, dependability, and adaptability requirements of the US Department of Defense. The Rainbow Project, conducted at Carnegie Mellon University, sought to provide basic architecture infrastructure for this approach, in addition to applying the technique to systems where performance was an important requirement.

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 [16,19]; Gorlick et al. use a dataflow style via Weaves [8]; and Magee et al. use bi-directional communication links via Darwin [11,12].

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 the research conducted under this grant, we demonstrated 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) they 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.

1.1. Innovative Claims

In our proposal, we suggested the development of new capabilities to reduce the cost and improve the reliability of making systematic changes to complex systems. The technology developed with the support of this grant enables significant improvements in our ability to:

1. **Detect dynamic (run-time) properties of complex, distributed systems.**

   A monitoring infrastructure consisting of a set of probes that collects status and performance information for networks and endpoints was developed. The information includes both static information that is useful for configuration-time adaptation and dynamic information that can be used to guide adaptation decisions at run-time.

   A new set of mechanisms, termed gauges, aggregate the results of multiple probes into information that is directly relevant to architectural analysis. The Rainbow

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

project developed the DASADA Gauge Infrastructure, which provides a common basis for gauges developed by other DASADA researchers to communicate monitoring information about a running system to adaptors that dynamically adapt a software system. Additionally, Rainbow provided gauges and probes to report information about network performance, as well as a system called DiscoTect for discovering the runtime architecture of a target system.

2. **Determine whether those properties violate critical assumptions of a running system.**

   New run-time monitoring infrastructure allows a system to introspect about its own properties, determining when existing system behavior is inconsistent with expected operating assumptions and parameters. This permits the system to rapidly detect, before system failure or severe degradation, when it needs to be adjusted or reconfigured. The Rainbow project retargeted the Acme toolset to provide runtime architecture reporting, constraint analysis and detection of violations of architectural rules at runtime.

3. **Automate system adaptation and repair in response to violations of architectural assumptions**.

   "Repair strategies" associated with the architectural style permit the system to automatically adapt itself to certain classes of violation. In Rainbow, we provide a specific set of repair strategies tuned to performance enhancement (in combination with the performance gauges noted above). We also provide a repair engine, called Tailor, that can be used to execute repair strategies to change the architecture and propagate those repairs to the running system.

   In addition to on-line repairs, off-line repair actions are propagated into the system implementations using a new generation approach, termed "compositional connectors." Using it, one adapts interaction mechanisms by incrementally adding new capabilities to support changes in performance, security, or reliability.

The ability to introspect relies on a run-time representation of system architecture models and constraints over those models, building on the Acme Architecture Description Language (ADL) infrastructure developed under DARPA's Evolutionary Design of Complex Systems (EDCS) program. In addition to tools built specifically for this project, this infrastructure enables the integration of analysis tools developed under DASADA and EDCS, to detect constraint violation.

In combination, these capabilities radically improve the ability to (a) handle system changes with respect to the performance-oriented gauges supported by our technology, and (b) incorporate additional gauges and system adaptation rules produced by other DASADA-funded projects. This dramatically reduces the need for user intervention in adapting systems to achieve quality goals through reliable, architecture-driven self-adaptation.

We have evaluated the technologies in the context of distributed systems, which typically depend heavily on the performance properties of the run-time environment, and which exhibit considerable variability in their architectural requirements. Our demonstration testbed, built on standard network platforms and using standard

application components, allows us to benchmark our new capabilities for adaptation, and determine both the strengths and limitations of our mechanisms.

In Section 2 of this report, we describe the technical contributions of our work in terms of specifying architectures and using them for run time adaptation. In Section 3, we describe the tool support that we have developed for applying our approach. Case studies and evaluations are described in Section 4. Finally, in Section 5 we discuss our conclusions and areas of future research.

2. Approach and Framework

Our starting point is an architecture-based approach to self-adaptation, similar to [17] (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.

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 First In First Out (FIFO) order by the servers in the server group. Individual servers send their results directly to the requesting client.

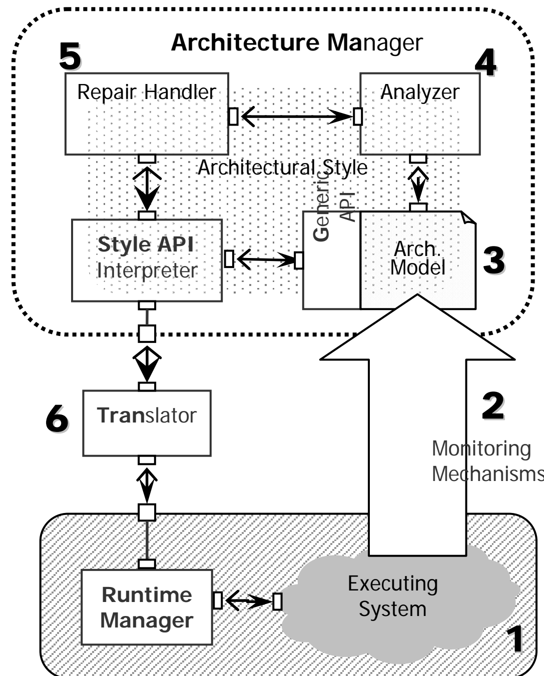


Figure 1. Adaptation Framework.

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

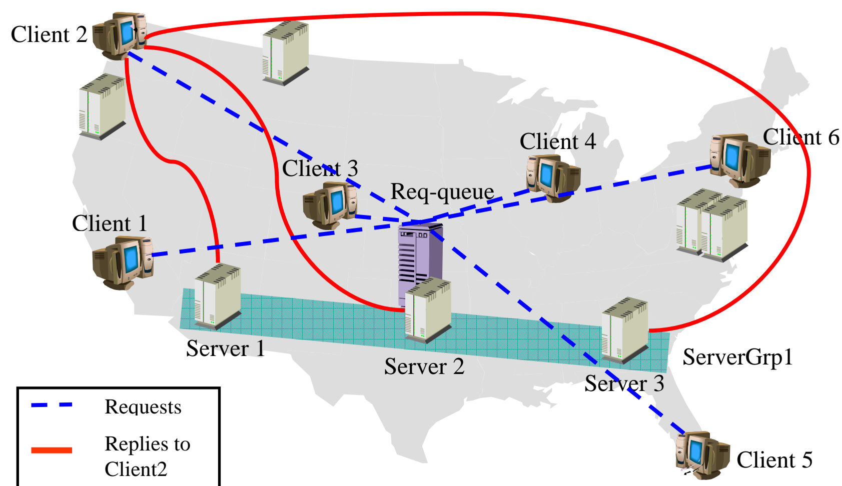


Figure 2. Deployment Architecture of Example System.

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. 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. 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 use existing off-the-shelf performance-oriented "system probes," using probes from other DASADA researchers (such as Active Interface Development Environment (AIDE) from George Heineman at Worcester Polytechnic Institute [9] and ProbeMeister from Dave Wells at Object Services and Consulting, Inc. [21]). 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. This translator can interact with Workflakes from Gail Kaiser at Columbia University (http://www.psl.cs.columbia.edu/old/WorkFlakes/) to effect changes in the actual system.

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.

In the remainder of this section, we discuss in more detail each aspect of the architectural adaptation framework.

2.1. Architectures and Architectural Style

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 [6, 18, 23]. Moreover, the notion of style often maps well to widely-used component integration infrastructures (such as Enterprise JavaBeans, High Level Architecture, Common Object Request Broker Architecture), 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 [19] supports a style based on hierarchical publish-subscribe; Wright [1, 2] supports a style based on formal specification of connector protocols; MetaH [20] 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

---

[2] This is the core architectural representation scheme adopted by a number of ADLs, including Acme [8], xArch [3], xADL [5], ADML [15], and SADL[14].

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 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 [7], a generic ADL that extends the above structural core framework with the notion of style. The rules and constraints are defined in Armani [13] a first-order predicate logic similar to the Unified Modeling Language's Object Constraint Language (OCL), 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]

---

[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.

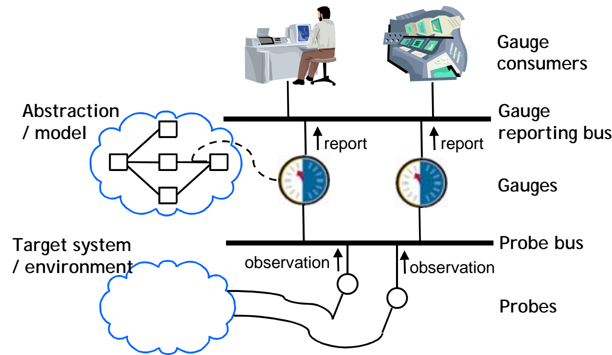


Figure 3. Gauge Infrastructure.

2.1.1. 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 [20], and Wright uses protocol model checking [1]. 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, 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.

2.2. 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 3. This monitoring infrastructure is described in more detail in Section 3: 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

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

"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 3 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.

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 [10]) 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 [21] to generate the actual probes, without writing any additional code.

2.3. Analysis

In order to determine if repair is needed, it is necessary to analyze the architecture in the context of monitored information. As described in the previous section, monitoring information is stored as properties in the architecture. Analysis in the Rainbow framework is conducted by evaluating architectural constraints represented in Armani [13]. Whenever a property value changes, Armani rules are re-evaluated; if the constraints fail, then repair strategies associated with the constraints are triggered. This is described in the next section.

2.4. Reconciliation

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.

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.

2.4.1. 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 [22]. 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 filter 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 [22] or [23].

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.

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.

2.4.2. 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 executes 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.

2.5. Propagation

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, or as complicated as mobile-code or an entire change propagation technology.

3. Tool support

In this section, we discuss the tool support that we developed as part of the DASADA program. We developed support for each of the parts of our infrastructure, in addition to development tools to aid in the development of particular gauges and repairs.

3.1. Gauge Infrastructure

To illustrate how the infrastructure is realized in practice consider Figure 4, which presents a simple example of probing and gauging. Imagine that we have a target system that consists of a *sender* that is sending 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 network 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. 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

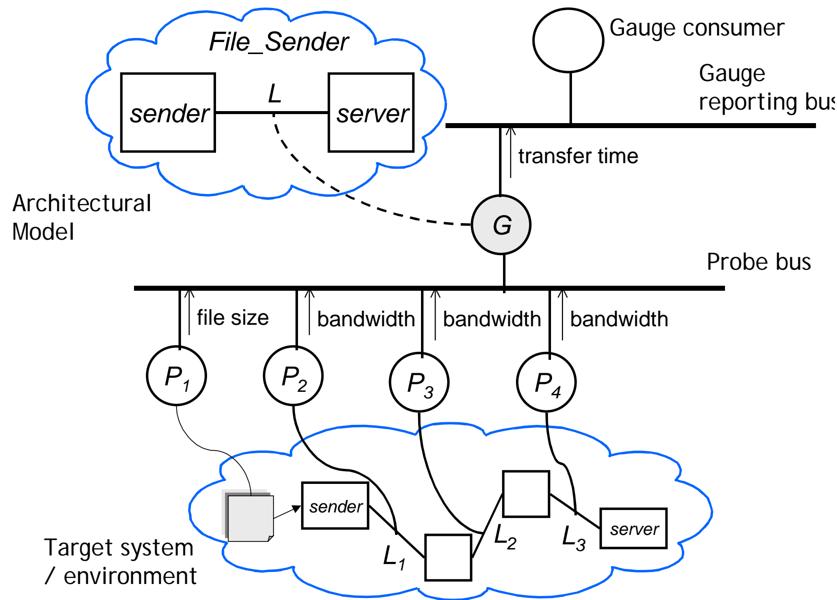


Figure 4. Gauge Example.

system with respect to this performance attribute, we need to insert some probes and gauges.

Two types of probes are inserted 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 might be realized through instrumenting the system call *fopen*, for example. 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 monitoring, a gauge is attached to the connector L in the architectural model. 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.

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

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, defined in Table 1, 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.[6]
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 4 that measures the transfer time value in milliseconds, represented as a floating point number.

Table 1. An Example of Gauge Type Specification.

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.`

How a given gauge type is instantiated is described in a gauge instance specification as 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 4, which is of gauge type *XferTime_Gauge_T*;

---

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

2. The name and type of the model that the gauge is associated with: for example, *G* is associated with a model called *File_Sender*, which is of type *Acme*;
3. Mappings from values reported by the gauge to the associated model properties. Each mapping is a tuple of <*GaugeValue, ModelProperty*>, meaning that the *GaugeValue* actually reflects the value of *ModelProperty*: 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.

Table 2. An Example of Gauge Instance Specification.

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.`

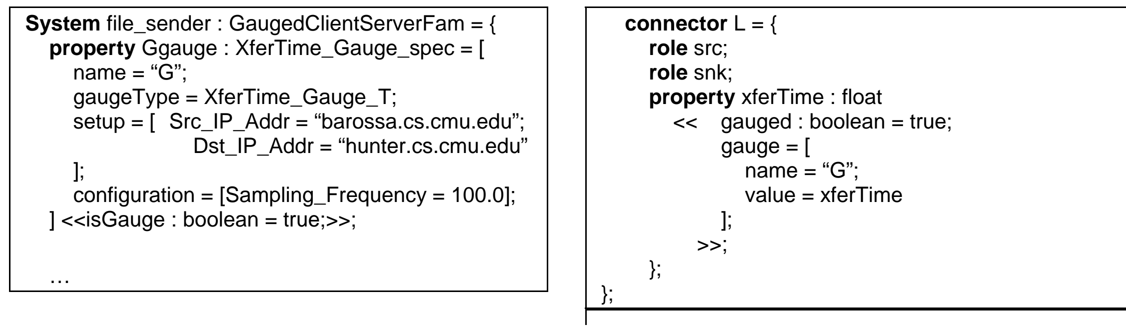
The above definition of gauges is very general, can be applied to a wide variety of monitoring needs, models, or modeling languages.

3.1.2. Implementation

To this point we have described generally the way that 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 the state of our implementation.

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


```
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 5. Attaching Gauges to Systems in Acme.

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 other tools that analyze 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.

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 5 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.[7] Figure 5 also shows an Acme description of a connector *L*. 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*) defines the name and type of the gauge, which gauge value is mapped to this particular property, and the setup and configuration parameters. The Acme gauge specification in Figure 5 corresponds to the gauge instance specification in Table 2. The *gauge* meta-property is an Acme record that is defined elsewhere in the Acme description. Each gauge type has a corresponding Acme record type. These records can be generated automatically from the gauge type specification.

---

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

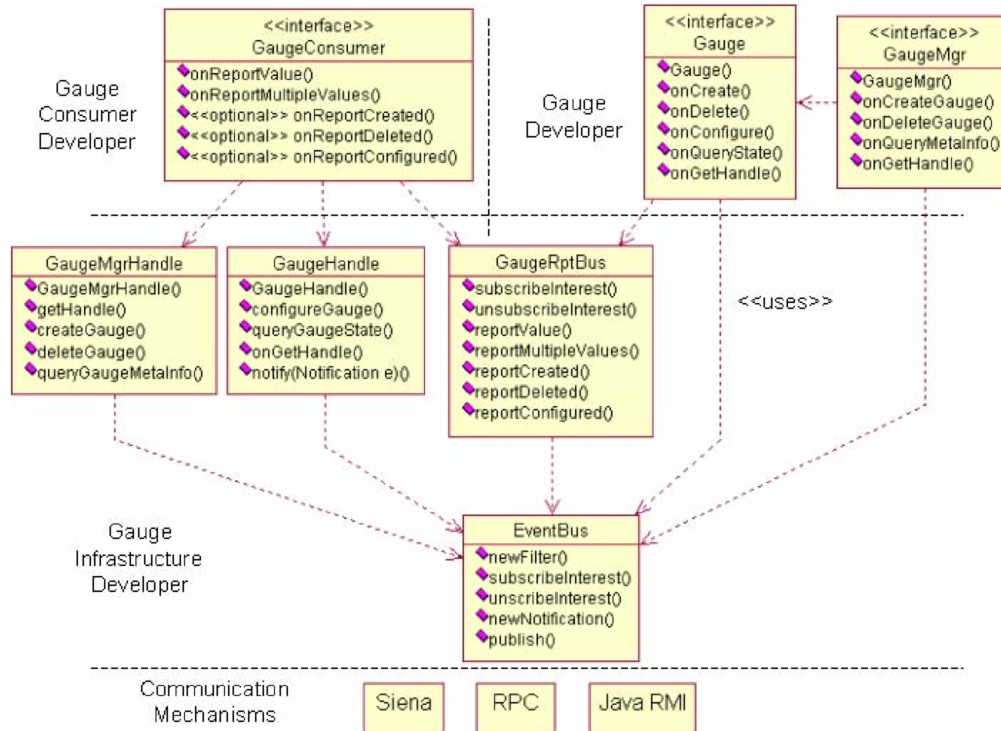


Figure 6. Gauge Infrastructure Implementation.

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 uses the Siena wide-area event notification system [3] as the communication substrate through which events are communicated between gauges and their consumers. The class hierarchy for this implementation is presented in Figure 6. 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 Siena, and another that transparently utilizes Java Remote Method Invocation (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.

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.[8] Figure 7 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

[8] This tool is currently being implemented and will be ready by the time camera-ready copies of the paper are due.

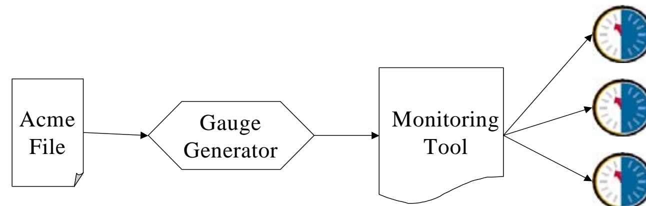


Figure 7. Generating Gauges from Acme Descriptions.

Common Object Model (COM) interface of AcmeStudio to load the Acme description, start listening for gauge values, and propagate these values to AcmeStudio.

3.1.3. Gauge Workbench

The gauge specifier is a Java application for specifying gauge types. Gauge types define the values reported by instances of these gauges, the parameters required to create a gauge (called the setup parameters), and the parameters that may be used to configure instances as they run. The gauge specifier provides a GUI front end to this specification. The output of the specifier is:

- **Acme Families:** that define the gauge type for use within AcmeStudio. This allows gauge instances to be attached to Acme designs and created from within AcmeStudio.
- **Gauge Implementation Stubs:** Generate Java stubs for gauges and gauge managers that integrate with the CMU Gauge Infrastructure. The aim is that the gauge developer has to write the minimal amount of code to have a gauge implemented.

3.2. Gauges

Gauges interpret system-specific information in the context of an architectural model. During the period of this grant, we investigated three different types of gauges.

3.2.1. Network Performance Gauges

Network Performance Gauges measure characteristics of the network and report these as properties in an architectural model. We have developed probes for gathering information about networks, based on the Remos system [10]. Remos has two parts: 1) an API, which allows applications to issue queries about bandwidth and latency between groups of hosts, implemented as a library that is linked with applications; and 2) a set of servers, called *collectors*, that collect information about different parts of the network. A probe uses Remos to collect the information required for the probe (such as bandwidth and latency) and distributes it as events using the DASADA Probe Infrastructure Protocol.

Our performance gauges listen to this information and perform calculations and transformations to relate it to the architectural model of a system.

3.2.2. Protocol Gauges

In addition to providing information about the network performance, we have developed gauges that monitor the specified behavior of a system. These gauges take protocols specified in Finite State Processes (FSP) [11] and then monitor method calls in the system to see if the protocol is being followed in the running system. Protocols are specified as properties in an architectural model; a tool extracts these protocols and feeds them into gauges that interpret the protocol. The protocol gauges listen to probes that report method invocations in the running system and relate them to events in the protocol. The gauges then report the success or failure of the protocol to the architectural model.

3.2.3. Architecture Gauges

The Rainbow implementation currently assumes that the architectural model is consistent with the system. This is not necessarily the case – and, in fact, the architectural model of a system may not even be reliably known. In order to address this, we have developed technology, called DiscoTect, to monitor a running system and extract its architecture. The technology essentially uses state machines to monitor events in the running system and emit architectural events to create an architectural model. Details of this work can be found in [24].

3.3. Repair

In addition to providing tools to specify gauges and implement monitoring of a system, we have developed some infrastructure to handle and specify repairs. The Tailor repair engine is infrastructure that can be called when a constraint fails, and interprets repair strategies to determine the repairs that should take place. Currently, Tailor uses a simple scheme for interpreting repairs in a linear fashion; future research will look at making Tailor more intelligent to include learning which repairs have a history of working, for example.

In order to specify repairs, we have designed a language (examples of which appear in Section 4.1), that allows a designer to write the architectural repairs. Currently, we hand-translate these repairs into Java code that we can plug into the infrastructure.

3.4. Integration with Acme Tools

So far, we have discussed the tools that have been used for architectural design, and given details of some design-time Acme tools that are used to construct and analyze architectures. If we are using software architectural models and analyses to guide dynamic adaptation, then it is useful to use these tools at runtime. This approach preserves continuity between design time and runtime views of the system, and maintains uniformity of the types of analyses that are performed at runtime and their meaning with respect to the design-time architectural artifacts.

Given that we want to use existing architectural tools at runtime, the question arises as to what role they should play in runtime adaptation, how they should be adapted to be

used in the dynamic context, and what additional tools are required. The guiding principle should be to maintain the separation of concerns that exist in the framework outlined earlier, thus separating the different kinds of expertise required into different appropriate tools, rather than attempting to develop a monolithic tool to perform all aspects of adaptation.

While we discuss this with respect to some Acme-based architecture tools, we believe that analogous modifications will need to be made to any architecture tool to fit into the general adaptation framework of Figure 1.

Within the framework, the different separations of concern are:

- the use of different architectural views and analyses, that may in fact exist in several design-time tools;
- the ability to monitor different attributes of the architecture at different times in execution;
- the desire to experiment with different types of repair strategies in the framework; and
- the fact that there may be many mappings from a particular architectural model expressed in a particular architectural style to an implementation of that system.

Thus, in the design of our toolset we have modified our existing design-time tools to observe and analyze the architecture, and developed new tools to capture the knowledge particular to each concern.

3.4.1. Changes to Existing Tools

The changes to our existing toolset fall broadly into the following categories:

- Interfaces that allow the architectural model to be changed dynamically.
- Integration points between architectural analysis tools and facilities to effect a repair should analysis determine something is wrong.
- Facilities to allow a designer to indicate points in the architecture that should be monitored, and the types of monitoring that should be conducted.
- Addressing the scalability issues with conducting analysis at runtime, in reaction to observations of the executing system.

We show how we addressed these categories in the case of AcmeStudio and Armani.

AcmeStudio: The role of AcmeStudio in the dynamic adaptation framework is twofold. First, it is still used at design time to define the architecture. For this stage, AcmeStudio has been extended to allow gauges to be attached to points in the architecture. Once again, this is based on families – families define which gauge types are available to a system. If a family defines such gauge types, instances can be dropped onto the design and attached to properties in the architecture. An external tool is called by AcmeStudio to start the gauges and begin reporting values.

The second role of AcmeStudio is as an observation tool in the adaptation framework. Once the system is started, AcmeStudio is no longer used to edit the architecture – in fact, it observes the changes made to the architecture by other tools. To facilitate this role, AcmeStudio has been extended with a COM interface through which gauges can report changing property values. The COM interface also contains routines to change the architectural model – create, delete, or modify components, connectors, etc. In this way, tools that do the actual analysis and modification can inform AcmeStudio, so that the changed architecture can be viewed. For example, if a gauge detects an overloaded server, it can report this fact as the sOverloaded property of the corresponding architectural component. AcmeStudio, using existing visual variants, can change the component to light gray.

Armani Constraint Analysis: Armani has been extended with an imperative language that can be used to define repair strategies to programmatically change the architecture. A *repair strategy* is associated with an Armani constraint, and is invoked when the constraint fails. A repair strategy is composed of a number of subsidiary constraints and repair tactics. This allows a repair strategy to conduct more than one change, based on further investigation of the problem. For example, if an Armani constraint specifying that latency must be below a certain threshold is violated, the repair strategy will likely contain tactics to address the case if the bandwidth has fallen or the load on servers has risen. Furthermore, repair strategies contain decision logic for choosing which of the tactics to apply.

A repair strategy for this scenario is presented in Figure 12. The particular Armani constraint, and the particular repair strategy to invoke, are shown in lines 1-3 of the figure. In line 2, "!→" is a new operator that specifies that the repair strategy following is to be executed only if the constraint is violated. The top-level repair strategy in lines 5-17, fixLatency, consists of two tactics, only one of which is chosen to be executed by this repair strategy. 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 architecture 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.

In addition to extending the Armani language, we are investigating ways to optimize the performance of the constraint analysis at runtime with incremental approaches.

3.4.2. New Architecture Tools

The existing tools address the concerns of observation and analysis in our framework. However, they do not address how to implement monitoring, how to execute the repair, or how to map between an architecture and its implementation.

Gauge Infrastructure: Gauges are used to propagate information about the runtime system to the architectural model. We have developed a gauge infrastructure that provides a Java class library to provide implementation stubs for gauges, and to facilitate communication between gauges and tools that consume gauge outputs. Because of the requirement for working in distributed systems, we have implemented the transport layer

of the gauge infrastructure using both the Siena wide area event notification system from the University of Colorado [3].

Tailor Repair: In concert with the repair extension to Armani, we are developing tools that provide runtime execution of these repairs. The goal of Tailor is to execute repairs that return an erroneous architecture to one that conforms to its style and constraints. Tailor listens to gauges for values associated with the model it is trying to maintain. It then invokes Armani to check if any constraints are violated. If they are, it executes the appropriate repair tactics. Tailor is decoupled from the executing system, and indeed can run on a machine independent of the running system. In this way, we anticipate that monitoring and repair at the architectural level will not unduly impede the running system.

Mapping Between Architecture and Implementation: Currently in our toolset we assume that gauges provide a mapping between runtime observations and architectural observations. In fact, this is just one example of mapping that is required throughout the framework. For our approach to be effective, we require a two-way mapping between information in the runtime and information in the architecture. Both directions are required by Tailor. A mapping from the implementation to the architecture is required when Tailor investigates the state of the running system to determine the best tactic (for example Tailor may need to determine which server group to move a client to). The mapping from the architecture to the runtime system is required when Tailor issues architectural changes that need to be reflected in the implementation. For example, Tailor may issue the architectural repair to add a Server component, which needs to be translated to starting a server process on a particular host and joining a particular server group. We are not assuming that the architecture to implementation mapping is one-to-one. Indeed, a particular architectural style, for example a client server architecture, could be associated with many "implementation styles." Currently, this information is captured in the Translator component of our framework and we are investigating methods of generalizing this component so that we can specify the transformations for multiple styles. Once this component is in place, it could also be used by gauges to associate runtime observations with architectural properties, in contrast to our current implementation, which embeds this information in the gauges themselves.

3.4.3. Integrating Architectural Tools

The development of different tools to capture specific knowledge about different aspects of dynamic adaptation means that these tools need to be integrated in some fashion. The framework in Figure 1 gives a broad outline of how to do this.

Figure 7 provides an illustrative example of how we have integrated our tools, in particular when applied to adapting a client-server system. The running distributed client-server system is on the left of the figure, and consists of three clients, three servers, and a request queue component. Clients make requests to the request queue and servers serve requests that they pull from the request queue. To instrument this system, each component is run inside an AIDE shell [9], which allows us to probe the method calls inside the component. This implementation corresponds to the example reported in Section 4, which calls for adaptation if the latency rises above 2 seconds. Using our tools

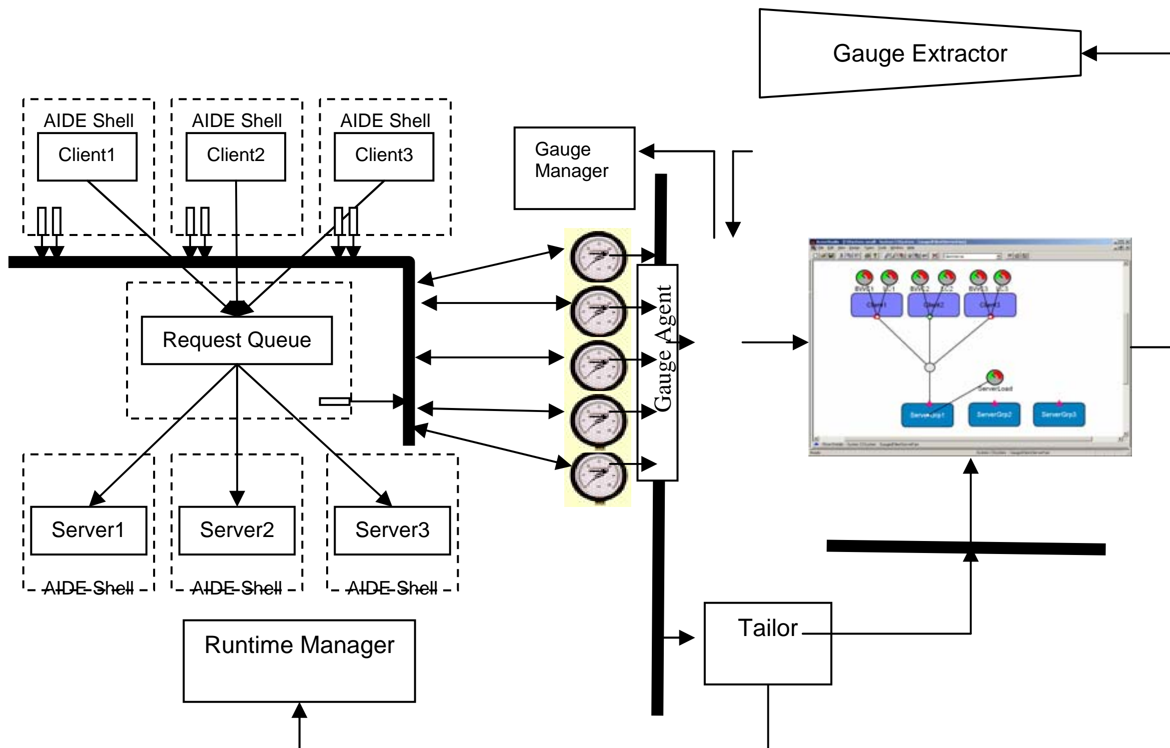


Figure 8. Integration of Architecture Tools.

to commence the adaptation requires several steps. It is assumed that the system is running, and that the architecture for this system is already defined.

1. AcmeStudio is used to attach gauges to various properties of the architecture. In our example, we attach gauges to the server load property of the request queue component of the architecture, and two gauges to each of the client roles in the architecture – one to report the bandwidth and one to report the average latency experienced by clients attached to the role.
2. To start the gauges, AcmeStudio invokes the *Gauge Extractor* tool, which communicates via RMI with a *Gauge Agent*. The Gauge Agent is the mediator between gauges and AcmeStudio.
3. The Gauge Agent locates *Gauge Managers* to start particular gauges and then creates the required gauges (in the middle of the figure).
4. These gauges create the necessary implementation probes. The probes in this example report every time the newRequest method is called in a client, and also the size of the response corresponding to the request. A probe in the Request Queue reports the size of the queue.
5. The gauges interpret this low-level, method-call information into high level latency and bandwidth values and report these values to the gauge bus.
6. The Gauge Agent reports gauge values to AcmeStudio, which can display the results.

7. Concurrently, Tailor listens to the gauge bus and evaluates Armani constraints to determine if the system is still performing acceptably. If not, it makes changes to its internal model of the architecture and reports these changes to AcmeStudio, via the COM interface, and the *Runtime Manager*, via RMI.
8. The Runtime Manager in this example contains a simple table-based mapping between architectural changes and runtime changes, and performs the necessary changes in the runtime based on the repair tactic chosen by Tailor.

4. Case Studies

4.1. Performance-based Adaptation of a Web-based Client-Server 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 2 to illustrate our technique. The example is simple load balancing of a web-based client-server system. This 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.

4.1.1. Defining a Client-Server Architectural Style

Figure 9 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).

```
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;
  };
};
```

Figure 9. Client/Server Style Definition.

```
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;
};
```

Figure 10. Client/Server Style Extended for Analysis.

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 10. 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 11.

4.1.2. 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, 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 3. 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*.)

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 [18] 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

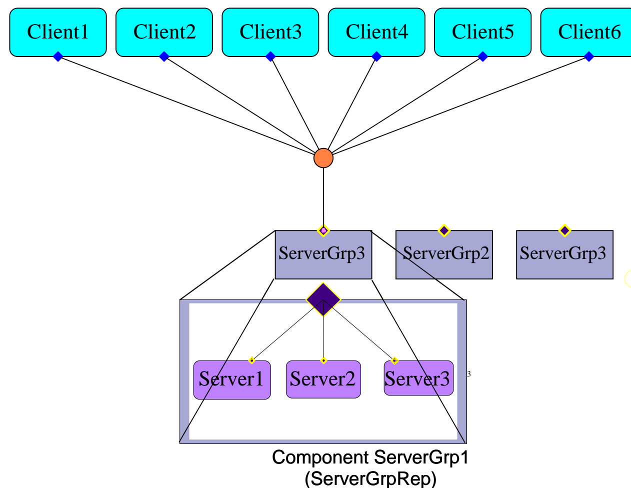


Figure 11. Architectural Model of Example System.

Table 3. 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)	$\sum T_s V_s + \sum D_c V_c$

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.

4.1.3. Defining Adaptation Operators

The client-server architectural style suggests a set of style-specific adaptation operators that change the architecture 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.

4.1.4. 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 3) 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

```
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  }
```

Figure 12. Repair Tactic for High Latency.

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 12 (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.

4.1.5. 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 PAClientRoleT in Figure 10). 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.

4.1.6. 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 4. 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 4.1.3 to the Runtime Manager operations using the scheme summarized in Table 4. (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.

4.1.7. 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 11 is transformed into the architecture in Figure 13.

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

Table 4. 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**

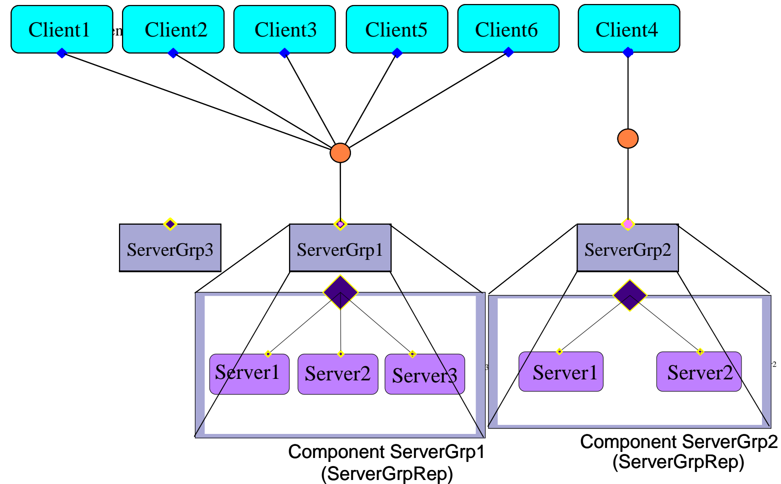


Figure 13. Model of System After Low Bandwidth Repair.

execute the repair strategy in Figure 12, passing Client4's role as a parameter.

6. The repair strategy first attempts to fix the server load, but returns false because no 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.

4.1.8. Results

We conducted an experiment to test the effectiveness of our adaptation framework on a system that has no built-in adaptation, and to elucidate the portions of our framework that needed more investigation.

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. We used this example because the architectural style of the system is amenable to automatic performance analysis [18], the results of which we can use to guide the development of our repairs.

This system is implemented in Java and has a set of change operations corresponding to the operations in Table 4, that are called via RMI to change the system. The clients send requests to an entity that splits the requests into queues, corresponding to the client's server group. Servers in a group pull information from the appropriate queue, and send a reply. The size of the reply is indicated by the client request.

The requirements and assumptions that fed into our analysis are:

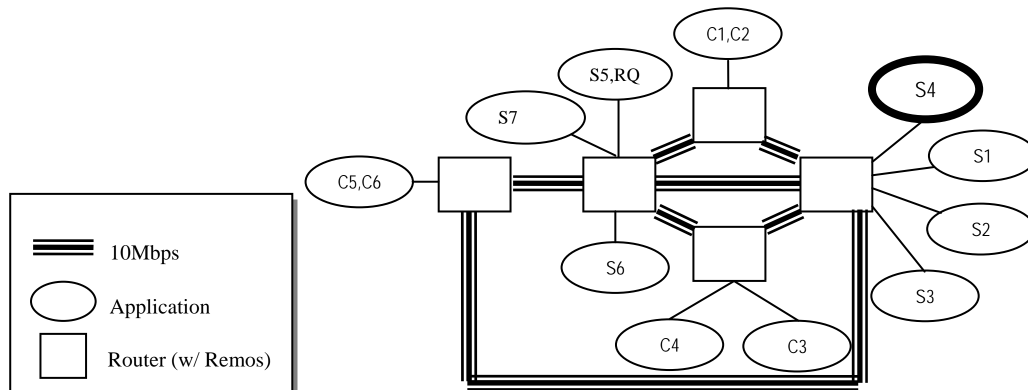


Figure 14. The Experimental Testbed.

- We desire the maximum average latency experienced by clients to be less than 2 seconds
- The size of client requests is small (0.5K on average) compared to server responses (20K on average).
- The average arrival rate of requests is approximately six per second.

Given these inputs, we calculated that an initial starting point of 3 replicated servers in one server group would be sufficient to serve our six clients, and that the bandwidth between the clients and servers should not be less than 10Kbps. Our experiment measured the effectiveness of our approach as compared to not using our approach.

Experimental Design

The experiment was conducted in a distributed setting inside a dedicated experimental testbed consisting of five routers and eleven machines (depicted in Figure 14), in which we deployed the client-server system. Because we had access to fewer machines than processes, Clients 1 and 2 (C1 and C2 in the figure) share a machine, and the request queue shares a machine with Server 5 (S5). In the initial state, Servers 4 and 7 were spare servers that we could activate as repairs warranted. The routers are connected via 10Mbps links; each application node is connected to a router by a connection that is at least 10Mbps. The repair infrastructure was restricted to the machine running Server 4 (the thick ellipse in Figure 14), except for those parts of the infrastructure associated with monitoring and communication of observations, which were distributed throughout the environment.

To measure the effectiveness of our approach, we examined how often the latency of any client exceeded two seconds, whether our repair was effective in reducing the latency to the required bounds, and how this compared with the latency experienced when our repairs were not conducted (the control). Because we used a network of machines, we were unable to eliminate all of the variables between the control and our experiment runs. However, we attempted to control as many variables as possible by: (1) seeding the clients so that the size of requests and responses occurred in the same sequence in both experiments, (2) executing a program that generates the same bandwidth competition for each experiment, and (3) isolating the network from outside traffic and users.

To ensure that repairs occurred, we needed to arrange the bandwidth competition so that there were periods of time where the bandwidth would cause the latency of some clients to be high. Similarly, the clients were controlled so that they requested larger amounts of information more frequently for a period of time. In this way, we ensured that there were periods of time during which the assumptions made in architectural performance analysis were invalid, and so that repairs were required.

The control and the experiment runs were executed under the conditions described above for a period of thirty minutes each. Figure 15 shows the stepping functions we used for generating bandwidth competition and server load. In the first two minutes, we ran the system in a quiescent state to give our gauges, probes, and system time to deploy and connect. In the following 8 minutes, we raised the bandwidth between the machines running Clients 3 and 4 (C3&4) and the machines representing Server Group 1 (SG1). In this period we would expect our repair strategies to migrate these clients to Server Group 2 (SG2). In the period 10 minutes to 20 minutes, we increased the server loads by increasing the file request size and rate of messages sent from all clients (20KB, twice every second), while reducing the bandwidth to SG1. In the final 10 minutes, we increased the bandwidth between C3&4 and SG2. During the periods of high bandwidth between C3&4 and their respective server groups, we maintained moderate bandwidth (3Mbps) between the opposite server groups. We needed to restrict the competition in this way because of the limited resources on our testbed. In future work, we plan to run the experiments with more realistic bandwidth data, based on network traffic to Carnegie Mellon's web server.

Results

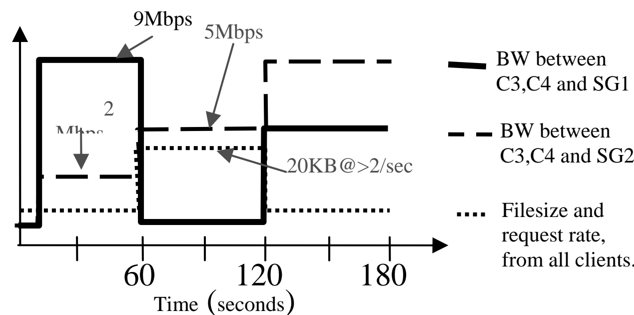


Figure 15. Bandwidth and Server Load Generation.

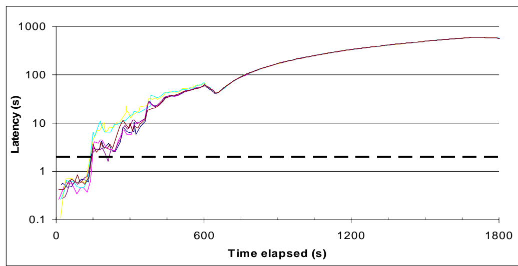


Figure 16. Average Latency for Control.

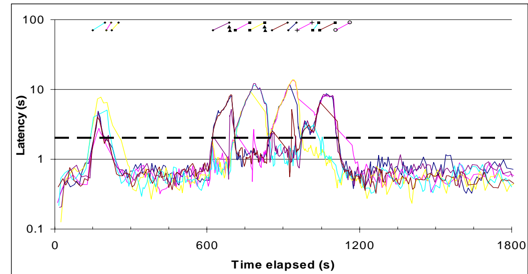


Figure 19. Average Latency under Repair.

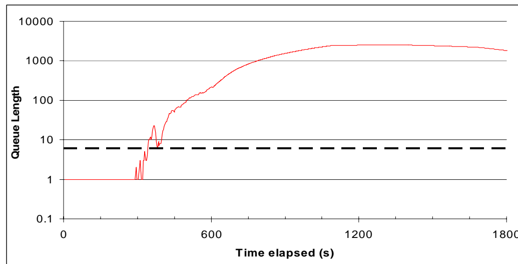


Figure 17. Server Load for Control.

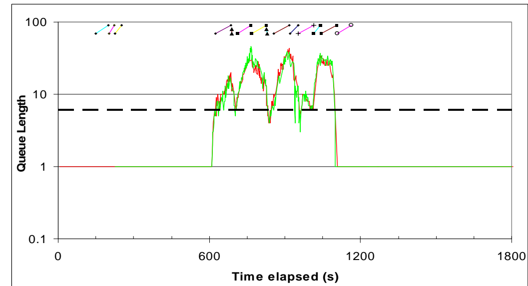


Figure 20. Server Load under Repair.

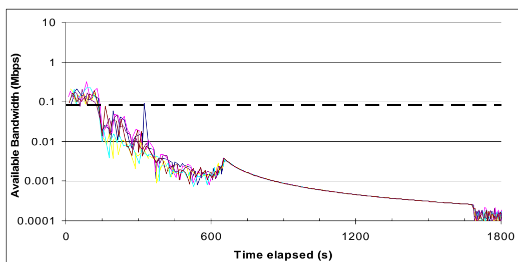


Figure 18.Available Bandwidth in Control.

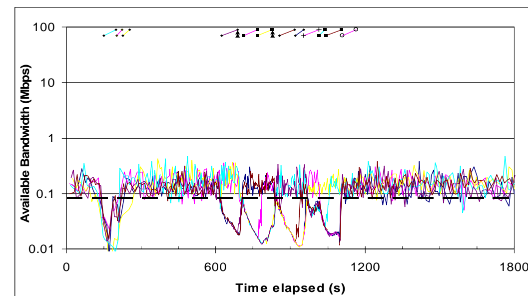


Figure 21.Available Bandwidth with Repair.

The results for the control run (without adaptation) are shown in Figure through Figure . The average latency, shown in Figure , continues to rise. Once the latency rises to above two seconds (at approximately 140 seconds for each client), it never falls below this required threshold. This is because the server load and bandwidth never recover. In Figure 17, the server load increases dramatically as the experiment progresses. (Note that we measure server load by measuring the size of the queue of waiting client requests.) Similarly, the available bandwidth falls dramatically as the experiment progresses, as shown in Figure . The dashed line in both figures indicates the limits that we used to decide which repair tactic to execute. In Figure 17, a queue size of greater than six waiting requests indicated that the server was overloaded, and so the server repair should be tried. In Figure , an available bandwidth of less than 10Kbps indicated that there was not enough bandwidth. Note that for the control run, we overloaded the system so much that it never recovers. However, toward the end of our run the servers actually begin to recover.

Figure 19 through Figure show the results obtained when our adaptation framework and repair strategies were applied under the same conditions as the control. Figure 19 shows a dramatic improvement in the average latencies experienced by the clients. Once our framework detects that client latency is above two seconds, a repair is invoked (either to move a client or add a server), and this improves the system performance as predicted by our design time analysis. In each of the figures for our experiment, the duration under which a repair is running is indicated by the lines at the top of the graph.[9] In fact, our framework has a positive effect on the available bandwidth because we are taking better advantage of different network links in our system after a repair. Our results for the server load show a marked improvement over the course of the experiment, except during the time that we increase the load on the server. During this time, we are continually performing repair. These repairs, encouragingly, do have a positive effect on the overall latency. Figure shows the server load experienced during the run. Note that the only time that the server load rises above the constrained value is when we stress the servers.

Discussion

The experiment indicates that the architectural approach improves the performance of the overall system, but further investigation is warranted under more realistic conditions. Repairs were conducted automatically by the system as needed, and the latency experienced by clients was less then two seconds for most of the time. In contrast, the latency experienced in the control spent a considerable amount of time over two seconds. When the system started to perform badly it continued to perform badly, and the indications were that it only started to recover toward the end of our control run.

As noted, during the period of increased server load, repairs are continually performed. Due to limited resources in our testbed, we were able to recruit only two extra servers. Once these were activated (at times 700 seconds and 800 seconds) the only repair possible was to move clients. During this period, we observed some oscillation, with clients moving back and forth between server groups. This movement still had a positive effect on the system, but we believe this is an artifact of the way we stressed the servers. Recall that the servers were stressed by sending large amounts of data more frequently. Of course, this also affects the bandwidth, and so the bandwidth repair does improve the system.

In running this experiment we found a number of areas on which to concentrate future work:

- The time that it takes to effect a repair averages 30 seconds. Most of this time is spent in communicating to create and delete gauges. Improving this time by caching gauges or relocating them (rather than destroying and creating new ones) should see our repair speed improve dramatically.
- The same network is being used to monitor the system as to run it. This means that when the available bandwidth is low, communication over our monitoring system is correspondingly slow. This produces a lag in the time when the bandwidth actually rises and the time it is noticed and repaired by our system.

[9] The gradient in these lines merely clarifies the beginning and end of a repair.

One way to address this is to use network Quality of Service (QoS) techniques to prioritize monitoring traffic.

- It is important to understand the underlying probe technology. The first Remos query for information about bandwidth between two nodes on the network takes several minutes because Remos needs to collect and analyze data. After this initial delay, the query is quite fast. To reduce this effect, we pre-queried Remos so that subsequent queries were much faster. Again, this reduced the time of our repairs. In general, this points to the need for more sophisticated probe technologies that need to be provided for caching or pre-fetching this information.
- In some instances, the effects of a repair on a system will take time. For example, adding a new server to a server group will not immediately reduce the overall load on the server group. Without taking this effect into account, unnecessary repairs are likely to occur (for example, to continue adding servers or to move clients). This type of delay is something that can only be gleaned from experience of running the repairs, and points to the need for a more sophisticated repair engine that can monitor repairs and their effects, and use this to adapt its repair policy.

Although we do not expect our approach to compete with hand-tailored, per-application adaptation, we believe that this approach will save time in engineering adaptation into applications that require it but do not possess it, in analyzing those repairs, and in changing them as required. However, this would be moot if the repairs did not improve the situation. These results show that we do get improvement by applying our framework – how this improvement compares to hand-tailored adaptation is an area of future work.

4.2. Performance Adaptation of GeoWorlds

In addition to the detailed case study described above, we have collaborated with Columbia University and Information Sciences Institute to apply our technology to provide load-balancing for GeoWorlds execution scripts. This is because a number of these scripts rely on computationally-intensive services, and these scripts needed to be made more resilient to service crashes and performance bottlenecks. Using probes developed by Columbia for monitoring GeoWorlds, we developed gauges attached to an architectural model of GeoWorlds that specified load constraints on the services. During execution of services, if the Rainbow infrastructure detected that a service load exceeded a threshold specified in the architectural model, Rainbow would conduct an architectural repair. This architectural repair was then translated into system-level repairs (carried out as workflakes [Workflakes]) on the GeoWorlds system.

5. Conclusions and Future Work

In this report, we outlined our research to generalize architecture-based dynamic adaptation to enable significant improvement in our ability to detect run time properties of complex, distributed systems, to determine whether those properties violate critical assumptions of a running system, and to automate system adaptation and repair in

response to violations of architectural assumptions. We have shown how the notion of software architecture needs to be modified to make it available at run time by providing architecture operators and repair strategies. Furthermore, we have demonstrated the effectiveness of this approach on a client-server example system and with a real-world military intelligence system, GeoWorlds.

Much of the foundational work for the science behind this approach, in addition to tool support to implement this science, has been conducted as part of the DASADA program. However, this research points the way to additional future work that could be carried out, for example:

- *Develop methodologies and tool support for dynamically determining the architecture of a running system.* The implementation of our approach has so far assumed that the architecture of the system is known. We detail the beginnings of research to dynamically detect architectures in [24].
- *Provide tool support for specifying architectural repairs*. Currently, we have a design for a repair language that can be used to specify strategies for repairing a system. In our implementations so far, we have hand-translated these into an associated implementation. We are investigating tool support, integrated with our architecture tools, to allow a designer to specify repair strategies in our repair language.
- *Investigate smarter repair engines.* Our implementation of Tailor provides simple support for executing repairs. In combination with the future work mentioned above, in addition to further research in planning and learning, it will be feasible to modify Tailor so that it has a more flexible means for determining which repair tactic to execute, it is able to detect whether a repair strategy is effected in the running system, and to provide some history and analyses of which repairs have been most effective in past repairs of the system.

6. Publications

- "DiscoTect: A System for Discovering Architectures from Running Systems," Hong Yan, David Garlan, and Bradley Schmerl. Accepted at the 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004.
- "AcmeStudio: Supporting Style-Centered Architecture Development," Bradley Schmerl, and David Garlan. Accepted at the 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004.
- "Increasing System Dependability through Architecture-based Self-repair," David Garlan, Shang-Wen Cheng, and Bradley Schmerl, in Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.
- "A Compositional Formalization of Connector Wrappers," Bridget Spitznagel, and David Garlan, The 2003 International Conference on Software Engineering (ICSE'03), Portland, Oregon, USA, May 3 - 10, 2003.

- "Software Architecture-based Adaptation for Grid Computing," Shang-Wen Cheng, David Garlan, Bradley Schmerl, Peter Steenkiste, and Ningning Hu, The 11th IEEE Conference on High Performance Distributed Computing (HPDC'02), Edinburgh, Scotland, July 2002.
- "Using Architectural Style as a Basis for Self-repair," Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, and Peter Steenkiste, *Software Architecture: System Design, Development, and Maintenance* (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture) Jan Bosch, Morven Gentleman, Christine Hofmeister, Juha Kuusela (Eds), Kluwer Academic Publishers, August 25-31, 2002. pp. 45-59.
- "Exploiting Architectural Design Knowledge to Support Self-repairing Systems," Bradley Schmerl, and David Garlan, The 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 15-19, 2002.
- "Reconciling the Needs of Architectural Description with Object-Modeling Notations," David Garlan, Andrew J. Kompanek, and Shang-Wen Cheng, Science of Computer Programming Volume 44, Elsevier Press, pp. 23-49.
- "Using Gauges for Architecture-Based Monitoring and Adaptation," David Garlan, Bradley Schmerl, and Jichuan Chang, In the *Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, 12-14 December, 2001.
- "A Compositional Approach for Constructing Connectors," Bridget Spitznagel, and David Garlan, *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Royal Netherlands Academy of Arts and Sciences Amsterdam , The Netherlands, August 28-31, 2001.

References

1. 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.
2. 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.
3. 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.
4. Dashofy, E., Garlan, D., van der Hoek, A., and Schmerl, B. http://www.ics.uci.edu/pub/arch/xarch/.

5. 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.
6. 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.
7. 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.
8. 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.
9. Heineman, G. Adaptation and Software Architecture. Proc. 3rd Annual International Workshop on Software Architecture (ISAW-3), pages 61-64, November 1998. Orlando, Florida.
10. 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.
11. Magee, J., and Kramer, J. Concurrency: State Models and Java Programs. Wiley, 1999.
12. 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.
13. Monroe, R.T. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.
14. 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.
15. The OpenGroup. Architecture Description Markup Language (ADML) Version 1. Apr. 2000. Available at http://www.opengroup.org/publications/catalog/i901.htm.
16. 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.
17. 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.
18. Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proc. the 1998 Conference on Software Engineering and Knowledge Engineering, Jun. 1998.

19. 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.

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

21. 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.

22. 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.

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

24. Yan, H., Garlan, D., and Schmerl, B. DiscoTect: A System for Discovering Architectures from Running Systems. Proc. 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004.