10-2004

Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure

David Garlan
Carnegie Mellon University, garlan@cs.cmu.edu

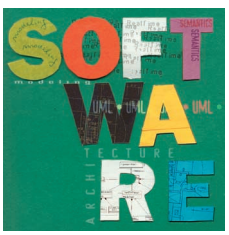
Shang-Wen Cheng
Carnegie Mellon University

An-Cheng Huang
Carnegie Mellon University

Bradley Schmerl
Carnegie Mellon University, schmerl@cs.cmu.edu

Peter Steenkiste
Carnegie Mellon University, prs@cs.cmu.edu

Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure



The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns.

David Garlan
Shang-Wen Cheng
An-Cheng Huang
Bradley Schmerl
Peter Steenkiste
Carnegie Mellon University

Software-based systems today increasingly operate in changing environments with variable user needs, resulting in the continued increase in administrative overhead for managing these systems. To reduce these costs, systems are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. Mechanisms that support self-adaptation currently exist in the form of programming language features such as exceptions and in algorithms such as fault-tolerant protocols. But these mechanisms are often highly specific to the application and tightly bound to the code. As a result, self-adaptation in today's systems is costly to build, difficult to modify, and usually provides only localized treatment of system faults.

In contrast to these internal mechanisms, recent work uses external models and mechanisms in a closed-loop control fashion to achieve various goals by monitoring and adapting system behavior at runtime.[1,2] As illustrated in Figure 1, control of system adaptation becomes the responsibility of components outside the system that is being adapted.

In principle, external control mechanisms provide a more effective engineering solution than internal mechanisms for self-adaptation because they localize the concerns of problem detection and resolution in separable modules that can be analyzed, modified, extended, and reused across different systems. Additionally, developers can use this approach to add self-adaptation to legacy systems for which the source code may not be available.

This external approach requires using an appropriate model to reason about the system's dynamic behavior. Several researchers have proposed using architectural models,[3] which represent the system as a gross composition of components, their interconnections, and their properties of interest.[4] Such an *architecture-based self-adaptation* approach offers many benefits. Most significantly, an abstract architectural model can provide a global perspective of the system and expose important system-level properties and integrity constraints.

While attractive in principle, architecture-based self-adaptation raises a number of research and engineering challenges. First, the ability to handle a wide variety of systems must be addressed. Since different systems have radically different architectural styles, properties of interest, and mechanisms supporting dynamic modification, it is critical that the architectural control model and modification strategies be tailored to the specific system. Second, the need to reduce costs in adding external control to a system must be addressed. Creating the monitoring, modeling, and problem-detection mecha-

Published by the IEEE Computer Society

nisms from scratch for each new system would render the approach prohibitively expensive.

Our Rainbow framework attempts to address both problems. By adopting an architecture-based approach, it provides reusable infrastructure together with mechanisms for specializing that infrastructure to the needs of specific systems. These specialization mechanisms let the developer of self-adaptation capabilities choose what aspects of the system to model and monitor, what conditions should trigger adaptation, and how to adapt the system.

THE RAINBOW FRAMEWORK

Figure 2 shows the Rainbow framework's control loop for self-adaptation. Rainbow uses an abstract architectural model to monitor an executing system's runtime properties, evaluates the model for constraint violation, and—if a problem occurs—performs global- and module-level adaptations on the running system.

Software architectures

Rainbow adopts a standard view of software architecture that is typically used today at design time to characterize a system to be built. Specifically, an architecture is represented as a graph of interacting computational elements.[4] Nodes in the graph, called *components*, represent the system's principal computational elements and data stores, including clients, servers, databases, and user interfaces. Arcs, called *connectors*, represent the pathways for interaction between the components. Additionally, architectural elements may be annotated with various properties, such as expected throughputs, latencies, and protocols of interaction. Components themselves may represent complex systems, which are represented hierarchically as subarchitectures.

However, unlike traditional uses of software architecture as strictly a design-time artifact, Rainbow includes a system's architectural model in its runtime system. In particular, developers of self-adaptation capabilities use a system's software architectural model to monitor and reason about the system. Using a system's architecture as a control model for self-adaptation holds promise in several areas. As an abstract model, an architecture can provide a global perspective of the system and expose important system-level behaviors and properties. As a locus of high-level system design decisions, an architectural model can make a system's topological and behavioral constraints explicit, establishing an envelope of allowed changes and helping to ensure the validity of a change.

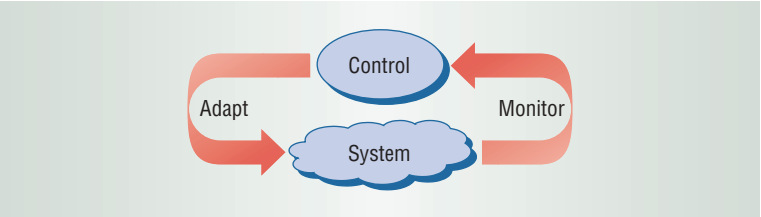


Figure 1. External control of self-adaptation uses external models to monitor and modify a system dynamically.

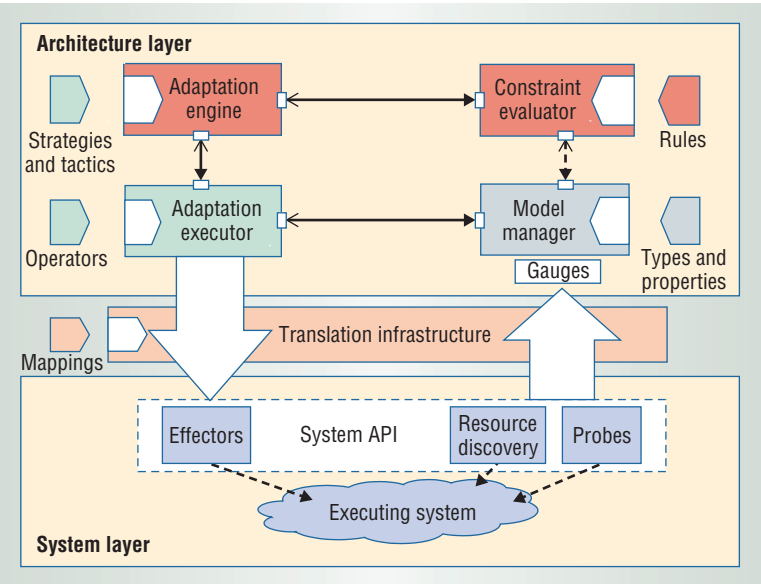


Figure 2. Rainbow framework. The framework uses an abstract model to monitor an executing system's runtime properties, evaluates the model for constraint violation, and—if a problem occurs—performs adaptations on the running system.

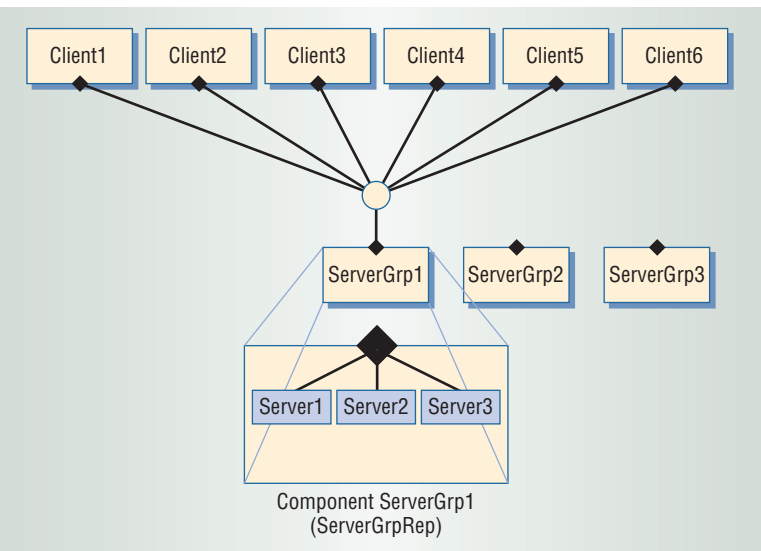


Figure 3. Client-server system software architecture. This model represents the architecture as a hierarchical graph of interacting components.

Figure 3 shows one example of an architecture in which the components represent Web clients and server clusters. Each server cluster has a subarchi-

To capture system commonalities, Rainbow adopts the notion of an *architectural style*, which describes a family of systems related by shared structural and semantic properties.

tecture consisting of one or more server components. This architectural model provides a global perspective on the system by revealing all the components and how they connect. The model also contains important properties such as each server's load, each connection's bandwidth, and the response time experienced by each client.

Further, the model maintains explicit constraints on the architecture that, for example, require each client to connect to exactly one server cluster. The constraints establish an envelope of allowed changes such as ensuring that no future changes to the system leave a client dangling without a connection. A system change is valid only if the system satisfies the constraints after the change.

Reusable Rainbow units

To fulfill Rainbow's objectives, its various components must be reusable from system to system. To identify what parts of the framework are reusable, and under what circumstances, we divide the framework into an adaptation infrastructure and the system-specific adaptation knowledge. The adaptation infrastructure, divided into system, architecture, and translation layers, provides common functionalities across self-adapting systems and is therefore reusable across all systems, while the adaptation knowledge itself is typically system-specific (Figure 2).

System-layer infrastructure. At this layer, we have defined the system access interface and built an infrastructure that implements it. A system measurement mechanism, realized as *probes*, observes and measures various system states. This low-level system information can be published by or queried from the probes. Additionally, a *resource discovery* mechanism can be queried for new resources based on resource type and other criteria. Finally, an *effector* mechanism carries out the actual system modification.

Architecture-layer infrastructure. At this layer, *gauges* aggregate information from the probes and update the appropriate properties in the architectural model. A *model manager* handles and provides access to the system's architectural model. A *constraint evaluator* checks the model periodically and triggers adaptation if a constraint violation occurs. An *adaptation engine* then determines the course of action and carries out the necessary adaptation.

Translation infrastructure. This infrastructure helps mediate the mapping of information across the abstraction gap from the system to the model and vice versa. A *translation repository* within the infrastructure maintains various mappings that the *translator components* share, for example, to translate an architectural-level element identifier into an IP address or an architectural-level change operator into system-level operations.

System-specific adaptation knowledge. Adding self-adaptation to a system using the functionalities that the adaptation infrastructure provides requires using the system-specific adaptation knowledge to tailor that infrastructure. This knowledge includes the target system's operational model, which defines parameters such as component types and properties, behavioral constraints, and adaptation strategies.

Architectural style

While reusable infrastructure helps reduce the costs of adding self-adaptation to systems, it is also possible to leverage commonalities in system architecture to encapsulate adaptation knowledge for various system classes.

To capture system commonalities, Rainbow adopts the notion of an *architectural style*. Traditionally, the software engineering community has used architectural styles to help encode and express system-specific knowledge.[5] An architectural style characterizes a family of systems related by shared structural and semantic properties. The style is typically defined by four sets of entities:

- *Component and connector types* provide a vocabulary of elements, including components such as Database, Client, Server, and Filter; connectors such as SQL, HTTP, RPC, and Pipe; and component and connector interfaces.
- *Constraints* determine the permitted composition of the elements instantiated from the types. For example, constraints might prohibit cycles in a particular pipe-filter style, or define a compositional pattern such as the starfish arrangement of a blackboard system or a compiler's pipelined decomposition.
- *Properties* are attributes of the component and connector types, and provide analytic, behavioral, or semantic information. For example, load and service time properties might be characteristic of servers in a performance-specific client-server style, while transfer-rate might be a property in a pipe-filter style.
- *Analyses* can be performed on systems built in an appropriate architectural style. Examples include performance analysis using queuing theory in a client-server system, and schedulability analysis for a real-time-oriented style.

These four entity sets primarily capture a system's static attributes. Rainbow extends this notion of architectural style to support runtime adaptation by also capturing the system's dynamic attributes, both in terms of the primitive operations that can be performed on the system to change it dynamically, and how the system can combine those operations to achieve some effect. Specifically, it augments the notion of architectural style with *adaptation operators* and *strategies*, which together determine the system's *adaptation style*.

- *Adaptation operators* determine a set of style-specific actions that the control infrastructure can perform on a system's elements to alter its configuration. For example, a service coalition style might define the operators AddService or RemoveService to add or remove services from a system configuration in this style.
- *Adaptation strategies* specify the adaptations that can be applied to move a system away from an undesirable condition. For example, a service-coalition system might have a system-wide cost constraint. Upon violating it, an adaptation strategy might progressively replace the most costly service with lower-grade services until the overall cost falls within acceptable bounds. Strategies are defined using—and therefore constrained by—operators and properties.

Although strategies use operators and properties to adapt systems of a particular style, they are designed for particular *system concerns*. A system concern outlines a related set of system requirements—such as performance, cost, or reliability—and determines the set of system properties on which self-adaptation should focus, and hence the set of strategies. The system concerns form a subset of the properties in a system's style. For example, a client-server system may have a style that includes load, bandwidth, and cost properties, while a particular performance concern might focus only on the system's load and bandwidth properties.

Adaptation style and system concerns together comprise two important dimensions of variability from system to system. How much of Rainbow's system-specific adaptation knowledge can be reused will depend on how similar two systems are in terms of their styles and system concerns. More specifically, types, properties, adaptation strategies, and operators may be reusable if the two systems have matching styles and concerns. Two case studies help illustrate this concept.

ADAPTATION CASE STUDIES

The following studies examine two systems with different adaptation styles that share the same system concern. This allows reusing the Rainbow framework across both prototype systems.

Rainbow supports runtime adaptation by also capturing the system's dynamic attributes.

Web-based client-server system

The first case study system consists of a set of Web clients, each of which makes stateless requests of contents from one of several Web server groups, as Figure 3 shows. The client and server components are implemented in Java and provide remote method invocation (RMI) interfaces for the effectors to use in performing adaptation operations. Clients connected to a server group send requests to the group's shared request queue, and servers that belong to the group grab requests from the queue.

The system concern focuses primarily on performance—specifically, the response time the clients experience. A queuing theory analysis of the system identifies that the server load and available bandwidth are two properties that affect the response time. Based on this system concern and analysis, the developer defines a client-server style for the system. The major parts of the style include

- ClientT, ServerT, ServerGroupT, and LinkT *types*;
- ClientT.responseTime, ServerT.load, ServerGroupT.load, and LinkT.bandwidth *properties*; and
- ServerGroupT.addServer() and ClientT.move (ServerGroupT, toGroup) *operators*.

The ServerGroupT.addServer() operator finds and adds an available ServerT to a ServerGroupT to increase the capacity. The ClientT.move (ServerGroupT, toGroup) operator disconnects ClientT from its current ServerGroupT, then connects ClientT to the toGroup ServerGroupT.

Associated with each client is an invariant that checks to see if its perceived response time is less than a predefined maximum response time. If the invariant fails, an adaptation strategy is invoked. An example invariant and adaptation strategy is

```
invariant (self.responseTime <
           maxResponseTime)
  !→ responseTimeStrategy(self);

strategy responseTimeStrategy
```

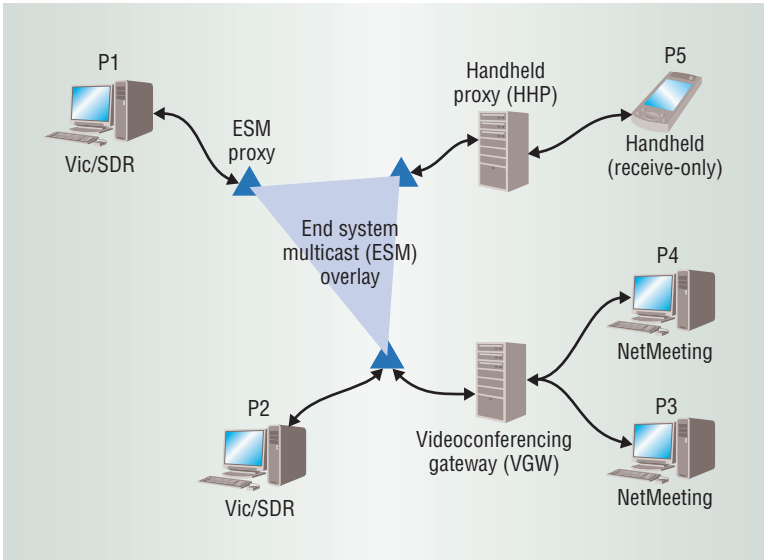


Figure 4. Videoconferencing adaptation case study. Two participants use Vic/SDR IP multicast videoconferencing tools; two others use NetMeeting, which adopts the H.323 protocol and unicast; while the final participant uses a handheld device running a slightly modified version of Vic.

```
            (ClientT C) {
let G = findConnectedServerGroup
            (C);
if (query("load", G) >
            maxServerLoad) {
      G.addServer();
      return true;
}
let conn = findConnector(C, G);
if (query("bandwidth", conn) <
            minBandwidth) {
      let G = findBestServerGroup
            (C);
      C.move(G);
      return true;
  }
return false;
}
```

In this specification, the invariant defines a predicate that determines whether a client's perceived response time (self.responseTime) is below a threshold (maxResponseTime). If this invariant is violated (indicated by "!→"), the adaptation engine executes the strategy responseTimeStrategy.

This strategy first checks to see if the current server group's load exceeds a predefined threshold. If so, the engine adds a server to the group to decrease the load and thus decrease response time. If, however, the available bandwidth between the client and the current server group drops too low, the engine moves the client to another group, resulting in higher available bandwidth and lower response time.

Videoconferencing system

Figure 4 shows the second system example, a videoconferencing session with five participating users. Two of the participants use the Vic/SDR videoconferencing tools, which use the Session Initiation Protocol (SIP) and IP multicast. Two other participants use NetMeeting, which uses the H.323 protocol and unicast. The final participant uses a handheld device, which runs a slightly modified version of Vic.

Since the handheld device cannot perform protocol negotiation, a handheld proxy (HHP) joins the conferencing session on behalf of the handheld user. A videoconferencing gateway (VGW) that supports both H.323 and SIP translates the protocols for NetMeeting and Vic users. Finally, to allow efficient communication among all participants across wide-area networks, the system uses Narada, an end-system multicast overlay consisting of three proxies, to provide the multicast functionality.

The system concerns here involve both performance and cost. For example, the system seeks to maintain sufficient available bandwidth between the handheld user and the handheld proxy, while keeping the cost of providing the videoconferencing service low. For example, if only one NetMeeting user remains online, the system should switch to a low-cost gateway. The developers define a videoconferencing style for the system based on these concerns. The major parts of the style include the following:

- VicT, NetMeetingT, HandheldT, GatewayT, HandheldProxyT, ESMProxyT, and ConnectionT *component and connector types*;
- GatewayT.cost, GatewayT.load, and ConnectionT.bandwidth *properties*; and
- HandheldT.move(HandheldProxyT, toHHP) and NetMeetingT.move(GatewayT, toVGW) *operators*.

In this case, the HandheldT.move(HandheldProxyT, toHHP) operator switches the handheld user to a new handheld proxy, while the NetMeetingT.move(GatewayT, toVGW) operator switches the NetMeeting user to a new video gateway.

Two sample adaptation strategies specify the desired adaptive behavior:

```
invariant (bandwidthToHHP (self)
          > minHHBandwidth)
  !→ HHBandwidthStrategy(self);

invariant (self.cost /
          numberOfNMusers <
          maxVGWUnitCost)
  !→ VGWCostStrategy(self);

strategy HHBandwidthStrategy
           (HandheldT HH) {
  let HHP1 = findBestHHP(HH);
  HH.move(HHP1);
  return true;
}

strategy VGWCostStrategy
           (GatewayT VGW) {
  let VGW1 = the gateway with the
           lowest cost that
           can handle the
           current load;
  if ((query("cost", VGW1) /
           numberofNMusers)
      < maxVGWUnitCost) {
    foreach NetMeeting user U of
           VGW {
      U.move(VGW1);
    }
    return true;
  }
  return false;
}
```

The first invariant is associated with components of type HandheldT, while the second invariant is associated with components of type GatewayT.

At runtime, when either invariant is violated, the adaptation engine executes the corresponding adaptation strategy. For example, when the available bandwidth between the handheld user and the handheld proxy drops too low, the engine moves the handheld user to a better handheld proxy.

When the unit cost of the gateway VGW becomes too high—for example, when a NetMeeting user leaves the session—the engine switches the NetMeeting users connected to VGW to the lowest-cost gateway that can handle the load.

A conflict between concerns of performance and cost is possible, such as when an adaptation pushes the system's total service cost above a maximum threshold. Although not addressed here, a composite utility function can help resolve such conflicts.

Reuse analysis

Several reuse issues can be better understood by examining how the two case study systems reuse the adaptation infrastructure's three layers and the various parts of style, which represent the system-specific adaptation knowledge.

System layer infrastructure. This layer consists of three elements: effectors, probes, and resource discovery. *Effectors* are component-specific: An effector can perform adaptation operations only on one type of system component. For example, a videoconferencing gateway effector can activate and shut down a gateway. Rainbow's system-layer infrastructure provides a reusable interface for accessing the effectors. For example, the adaptation engine/executor can issue an Activate operation by invoking the corresponding function that the system application programming interface (API) provides. The system- layer infrastructure then dispatches the operation to the appropriate effector based on the target component's type. Because the two case study systems do not share any components, their effectors cannot be reused.

The system-layer infrastructure provides *probes* that measure the response time, load, and bandwidth of various system components. Among these, the load and bandwidth probes are reused across the two systems because these two properties are of interest in both systems. Probes support monitoring and querying of information that is used at higher levels of the infrastructure to update model properties. In addition, the adaptation strategies often need to query some additional information, such as the server group load and new gateway's cost.

The adaptation engine needs a *resource discovery* mechanism to find available components to replace existing ones as directed by the adaptation strategy. For example, the first strategy for the videoconferencing system requires the adaptation engine to find the HHP component with the most available bandwidth for the handheld user. The second strategy finds a VGW component that supports both Vic users and NetMeeting users and has the lowest cost. Rainbow's system-layer infrastructure supports resource discovery based on component type and other desired component attributes.

Architecture-layer infrastructure. At the framework's architecture layer, the adaptation style spec-

At the framework's architecture layer, the adaptation style specifies the model instance's types, properties, and rules.

Table 1. Framework system-specific adaptation knowledge reuse summary.

Architectural style	System concerns	Reuse achieved
Different	Different	Adaptation infrastructure
Different	Same	Adaptation infrastructure, Properties, Mappings
Same	Different	Adaptation infrastructure, Types, Rules, Mappings, Adaptation operators
Same	Same	Adaptation infrastructure, Types, Rules, Properties, Mappings, Adaptation operators

ifies the types, properties, and rules of the model instance that the model manager handles. The functionalities of the gauges, model manager, constraint evaluator, and adaptation engine remain the same. Gauges for the properties of response time, load, and bandwidth aggregate information from the corresponding probes and update the appropriate properties in the model.

Translation infrastructure. This layer bridges the abstraction gap between the model and the system. For example, when the adaptation engine performs resource discovery to find a VGW, the translation layer must map the architectural type GatewayT to the system-level component type sysGateway that the system-layer resource discovery mechanism uses. Likewise, when the adaptation engine applies the connect operator to two elements in the architectural model, such as a NetMeeting user and a gateway, the translation layer must map them to actual machines in the system. These mappings are stored in the translation repository, and the translators perform the actual translation.

System-specific adaptation knowledge. The two case study systems differ in adaptation styles but share a system concern. This concern manifests itself in the properties of each style, so sharing the same concern means that the system can reuse the knowledge about the shared properties of load and bandwidth. Part of the knowledge is the translation mappings between the system properties and architectural properties that the style defines. For properties that the gauges need to aggregate, such as average latency, the aggregation knowledge can also be reused.

Drawing on these two case studies, Table 1 generalizes from our experience in determining what system-specific adaptation knowledge in the framework can be reused depending on the two dimensions of adaptation style and system concerns.

Implementation and evaluation

Moving beyond the two case studies, we have implemented a prototype of the Rainbow self-adaptation framework. At the system level, we use the global network positioning (GNP) approach to estimate network latency, the Remos tool[6] to measure bandwidth, and the network-sensitive service discovery (NSSD) mechanism[7] to discover resources. We implemented probes that obtain information from GNP and the Remos tool.

The architecture-layer entities are implemented in Java, based on the Acme architectural design toolset.[8] Performance-property gauges were implemented to read values from the probes and update the model manager's model via two event broadcast buses. For the translation infrastructure, the translation repository provides a Java RMI interface for the translators to use for storing and retrieving the necessary translation mappings.

Some translators are stand-alone entities, while others are integrated modules of system-layer or architecture-layer entities. Communications within the framework use XML messages over Java RMI.

Turning to the issue of reuse, although code size is not the only reuse measure, we can use the Rainbow prototype's code size to approximate the degree of reuse for the two systems. The Rainbow prototype—including the adaptation mechanism; model manager; gauge, probes, and their infrastructure; and the translation and system-layer infrastructure—requires 102 kilolines of code (KLoCs), with a breakdown of 84, 11, and 4 KLoCs for the architecture, system, and translation layers, respectively. Of these, the nonreused code and data for adaptation and translation mappings occupy about 1.8 KLoCs. In addition, the project reused 73 KLoCs of tool and utility code.

The Rainbow framework's self-adaptation effectiveness and performance are two other important aspects requiring evaluation. We can use the client-server system to demonstrate the effectiveness of the framework's self-adaptation. To demonstrate this, we conducted an experiment on a dedicated testbed consisting of five routers and 11 machines communicating over 10-megabits-per-second lines. This experiment conducted repairs, while the network was overloaded, on a client-server system that required a client latency of less than two seconds.

The results show that for this application and the specific loads used in the experiment, self-repair significantly improved system performance. Figure 5 shows sample results for system perfor-

mance with and without adaptation. Figure 5a shows that, without adaptation, once the latency experienced by each client rises above 2 seconds, it never again falls below this threshold. On the other hand, Figure 5b shows that if Rainbow issues the repairs, the client latencies return to optimal levels.

Perhaps not unexpectedly, our experiment also revealed that external repair has an associated latency. In the client-server example, it took several seconds for the system to notice a performance problem and several more seconds to fix it. For the videoconferencing application, elapsed time for adaptation at the architecture, translation, and system layers were 230, 300, and 1,600 ms, respectively, for one scenario, and 330, 900, and 1,500 ms, respectively, for another. Although we can imagine speeding up the round-trip repair time, these results indicate that the software architecture-based approach best suits repairs that operate on a systemwide scale and fix longer-term system behavior trends.

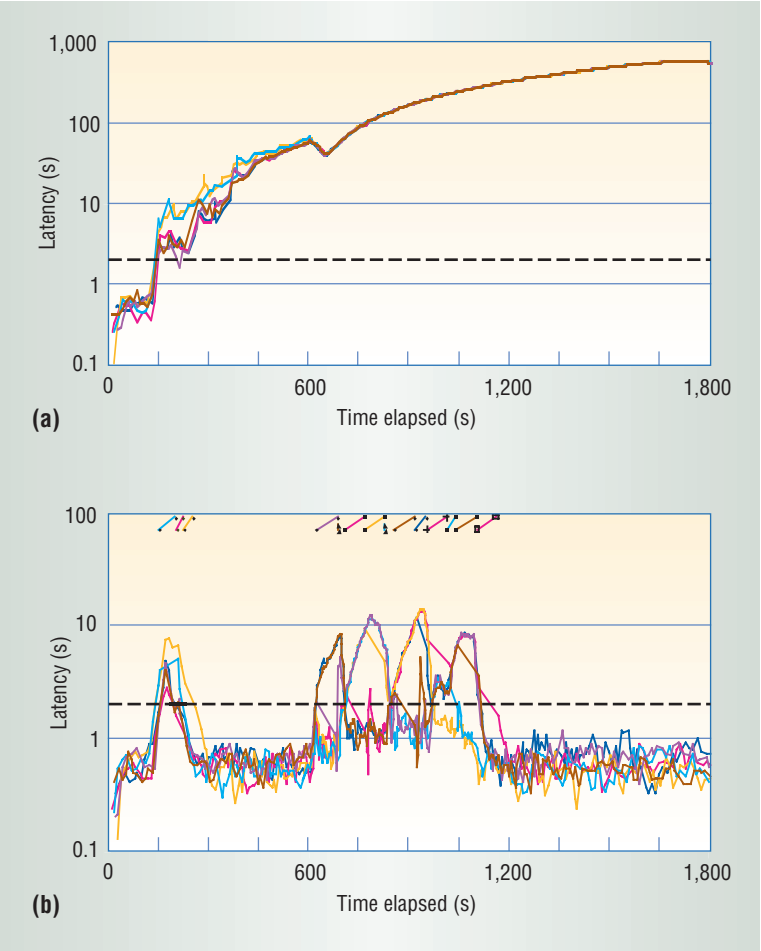


Figure 5. System performance with and without adaptation. The dashed lines indicate the desired latency behavior. (a) Without adaptation, if each client's latency rises above 2 seconds, it never again falls below that threshold. (b) Once repaired, client latencies soon return to optimal levels.

Although our evaluation indicates that it is possible to achieve architecture-based self-adaptation at low cost using reusable infrastructure, it is worth noting that the Rainbow framework rests on some important assumptions.

One assumption is that any target system will provide system access hooks for monitoring and adaptation. This assumption seems reasonable, at least for measurement, because an increasing number of measurement tools and infrastructure provide measures of or information about common component properties, including network bandwidth and latency. Several protocols can discover new services and resources for a system. Likewise, emerging effector technologies support dynamic changes to running system components. Also, developers can use wrappers to add hooks for making changes in legacy systems.

Another assumption lies in the adaptation infrastructure, where we assume probes will provide information about system properties, along with gauges to aggregate the information and update properties in the model manager. This assumption is based on a model that has well-defined gauge and probe APIs. We anticipate that external developers will specialize in developing gauges and probes for various purposes. Further, the measurement tools we have described could also implement the probe API or be wrapped to serve as probes, which would let developers of self-adapting systems plug in any gauge and probe to suit their needs.

Finally, the work we have described is inherently centralized, with monitoring and adaptation performed within a single Rainbow instance. Making this assumption has let us focus on core issues of self-adaptation—specifically monitoring, detection, resolution, and adaptation. At the same time, there may be concerns regarding scalability and single-point failure. The Rainbow framework can, however, be applied in a distributed setting. For example, we could apply Rainbow instances to adapt multiple subsystems of a distributed system, and then coordinate those instances toward an overall adaptation goal. The coordination and other distributed computing issues present a challenge for future research. ■

Acknowledgments

This research was supported by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, by the US Army Research Office (ARO) under grant number DAAD19-01-1-0485, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. The views and conclusions described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, NASA, the US government, or any other entity.

References

1. D. Garlan, J. Kramer, and A. Wolf, eds., *Proc. 1st ACM SIGSOFT Workshop on Self-Healing Systems* (WOSS 02), ACM Press, 2002.
2. A.G. Ganak and T.A. Corbi, "The Dawning of the Autonomic Computing Era, *IBM Systems J.*, vol. 42, no. 1, 2003, pp. 5-18.
3. P. Oriezy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, 1999, pp. 54-62.
4. P. Clements et al., *Documenting Software Architecture: Views and Beyond*, Addison-Wesley, 2003.
5. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
6. T. Gross et al., "Design, Implementation, and Evaluation of the Remos Network Monitoring System," *J. Grid Computing*, vol. 1, no. 1, 2003, pp. 75-93.
7. A.-C. Huang and P. Steenkiste, "Network-Sensitive Service Discovery," *J. Grid Computing*, vol. 1, no. 1, 2003; www.cs.cmu.edu/~pach.
8. D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Descriptions of Component-Based Systems," *Foundations of Component-Based Systems*, G.T. Leavens and M. Sitaraman, eds., Cambridge Univ. Press, 2000, pp. 47-68.

David Garlan *is a professor of computer science at Carnegie Mellon University. His research interests include software architectures, formal methods, self-healing systems, and task-based computing. He received a PhD in computer science from Carnegie Mellon University. Contact him at garlan@cs.cmu.edu.*

Shang-Wen Cheng *is a doctoral candidate at Carnegie Mellon University. His research interests include dynamic system adaptation, software architectures, and software designs for security. He received a BS in computer information sciences from Florida State University. Contact him at chengs@cmu.edu.*

An-Cheng Huang *is a doctoral candidate at Carnegie Mellon University. His research interests include distributed systems, networking, and grid computing. He received a BS in computer science and information engineering from National Taiwan University. Contact him at pach@cs.cmu.edu.*

Bradley Schmerl *is a systems scientist at Carnegie Mellon University. His research interests include dynamic adaptation, software architectures, and software engineering environments. He received a PhD in computer science from Flinders University in South Australia. Contact him at schmerl@cs.cmu.edu.*

Peter Steenkiste *is a professor of computer science and electrical and computer engineering at Carnegie Mellon University. His research interests include networking and distributed systems. He received a PhD in electrical engineering from Stanford University. Contact him at prs@cs.cmu.edu.*