Evaluating the Effectiveness of the Rainbow Self-Adaptive System

Shang-Wen Cheng, David Garlan, Bradley Schmerl
Carnegie Mellon University
{zensoul,garlan,schmerl}@cs.cmu.edu

Abstract

Rainbow is a framework for engineering a system with run-time, self-adaptive capabilities to monitor, detect, decide, and act on opportunities for system improvement. We applied Rainbow to a system, Znn.com, and evaluated its effectiveness to self-adapt on three levels: its effectiveness to maintain quality attribute in the face of changing conditions, run-time overheads of adaptation, and the engineering effort to use it to add self-adaptive capabilities to Znn.com. We make Znn.com and the associated evaluation tools available to the community so that other researchers can use it to evaluate their own systems and the community can compare different systems. In this paper, we report on our evaluation experience, reflect on some principles for benchmarking self-adaptive systems, and discuss the suitability of our evaluation tools for this purpose.

1. Introduction

Increasingly, systems have the requirement to self-adapt with minimal human oversight. They must cope with system errors, variable resources, and changing user priorities, while maintaining as best as they can the goals and properties envisioned by engineers and expected from users. However, self-adaptation in today's systems is costly to build, often taking many man-months to develop or retrofit systems with the capabilities. Moreover, once added, the capabilities are difficult to modify and usually provide only localized treatment of system errors.

We are investigating an approach that makes it possible for engineers to easily define adaptation policies that are global in nature and take into consideration business goals and quality attributes. In particular, we require that engineers be able to augment existing systems to be self-adaptive without needing to rewrite them from scratch, that self-adaptation policies can be reused across similar systems, that multiple sources of adaptation expertise can be synergistically combined, and that all of this can be done in ways that support maintainability, evolution, and analysis.

Our approach to self-adaptation uses architecture-based techniques combined with control and utility theories. Monitored properties of an executing system are reflected in an architecture model. The architecture model enables automatic reasoning about appropriate changes to improve quality-of-service in the target system. Utility theory is used to analyze tradeoffs across quality dimensions and select an appropriate adaptation strategy. Changes are then effected in the system, which is reobserved in a closed-loop form of control.

We evaluated Rainbow's effectiveness in adding self-adaptation to an existing system in two ways: (1) How effective was Rainbow at adapting a system to meet stated quality and business goals? (2) How much effort was required to engineer adaptation using Rainbow? To conduct the evaluation we designed and developed a web-based information system, Znn.com, to mimic real world systems, and an experimental environment to facilitate evaluation. In building Znn.com and making it amenable for evaluation, we developed tools to (1) monitor and effect changes on the Znn.com system, (2) provide interfaces for Rainbow to plug into; (3) produce an environment in which we could inject problems that need repairing; and (4) record the behavior of Znn.com both when self-adaptation is applied and when not. We were able to show that Rainbow was effective in adapting the system to meet stated goals, and that we could do so with considerably less effort than engineering a system to do the same from scratch.

Although numerous example and case-study systems have been proposed, researchers in the self-adaptive systems community generally lack a common benchmark to evaluate the effectiveness of their techniques and to compare their work to others'. Our evaluation experience yields a system and a suite of tools that can be used for such a purpose. We make the Znn.com system and associated tools available to the community as a benchmark tool suite.

In this paper, we describe how we evaluated Rainbow using the Znn.com experimental platform and re-

port our evaluation results. We then consider the requirements needed for a good benchmark environment that will allow comparison with other self-adaptive systems, and reflect on how our experimental platform meets those requirements.

2. Related work

IBM's Autonomic Computing initiative tackles challenges of emergent autonomic behavior with the MAPE control loop – to monitor, analyze, plan, and execute changes for self-management [16]. Their toolkit provides tools to diagnose problems and engineer autonomic systems [14]. Rainbow can be viewed as an instance of MAPE in which the shared Knowledge base consists of an explicit architecture model, a repertoire of adaptation strategies, and utility preferences.

To date, several dynamic software architectures and architecture-based adaptation frameworks have been proposed and developed [3],[22]. Related approaches focus on formalism and modeling, mechanisms of adaptation, or distribution and decentralization of control. These include Darwin with π-calculus semantics to specify distributed systems [17], ArchWare with architectural reflection and dynamic co-evolution [20], Weaves for construction and analysis of data-flow systems [9], Willow for survivable systems [27], ArchStudio for self-adaptation of C2 hierarchical publish-subscribe systems [5], Plastik targeting performance properties [2], CASA for resource availability concerns in mobile network environments [21], and CR-RIO for architectural reconfiguration using *contracts* [25]. These approaches share a few common characteristics: They generally apply closed-loop control and use an architecture model to reason about the target system. However, whereas most approaches assume certain structures in the target system and adapt for a fixed set of quality attributes, Rainbow is generic to architectural styles and handles multiple objectives. Surveys of existing self-adaptive systems, not limited to an architecture-based approach, can be found in [6] and [12].

Various autonomic systems have been evaluated in specific domains, e.g., database optimization [18], network server provisioning [23], and workload optimization in web servers [24]. Many of these evaluations focus on performance and overhead, but some also evaluate other criteria. McCann summarizes nine of these in [19]. Our evaluation of Rainbow focuses on quantifying success in meeting quality-of-service goals and the overhead of running the infrastructure, which are two of the criteria discussed by McCann. In addition, we consider the engineering cost of applying the framework to an existing system.

3. The Rainbow Approach

Rainbow [4],[7] focuses on two means of achieving cost-effective self-adaptation: an approach and mechanism to reduce engineering effort and an explicit representation of adaptation knowledge. It provides a framework of mechanisms to **monitor** a target system and its executing environment and reflect observations into an architecture model, **detect** opportunities for improvements, **decide** on a course of adaptation, and effect changes (**act**). Leveraging the notion of *architectural style* [1] to exploit commonality between systems, Rainbow provides general, reusable infrastructures with explicit customization points to apply it to a wide range of systems. It also provides useful abstractions to focus engineers on adaptation concerns, facilitating its systematic customization to particular systems. To automate system adaptation, it provides a language, Stitch, to represent routine human adaptation knowledge using high-level adaptation concepts of *strategies*, *tactics*, and *operators*.

3.1 Customizable Self-Adaptation Framework

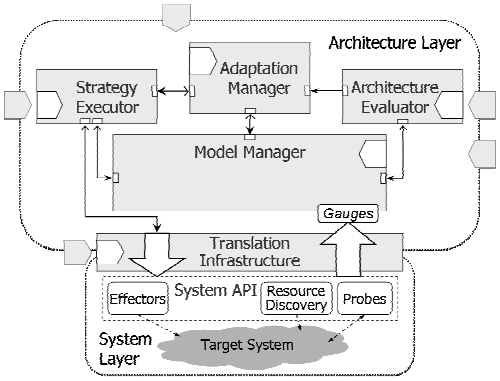

Figure 1. The Rainbow framework

The Rainbow framework (Figure 1) uses a component-and-connector architecture model of the target system to monitor the system and reason about appropriate strategies. The monitoring mechanisms in the Translation Infrastructure – probes and gauges – observe the running target system and update properties of an architecture model managed by the *Model Manager*. The *Architecture Evaluator* evaluates the model upon update to ensure that the system is operating within an acceptable range, as determined by the architectural constraints. If the Evaluator determines that the system is not operating within the accepted range, it triggers the *Adaptation Manager* to initiate the adaptation process. The Adaptation Manager chooses a suit-

able strategy based on current states of the system as reflected in the model. The *Strategy Executor* executes that strategy on the running system via system-level effectors. Rainbow is customizable to different domains: The architecture model of the target system customizes the Model Manager. Architectural constraints, related to business objectives to adapt for, customize the Architecture Evaluator. Style operators and their mappings to target-system effectors customize the Strategy Executor. Finally, utility preferences and a repertoire of strategies with their associated cost-benefit impacts customize the Adaptation Manager.

This customizable self-adaptation framework has a number of advantages. Providing a substantial base of reusable infrastructure greatly reduces the cost of development. Customization mechanisms allow engineers to tailor the framework to different systems with relatively small increments of effort. In particular, the tailorable model management and adaptation mechanisms give engineers the ability to customize adaptation to address different properties and quality concerns, and to add and evolve adaptation capabilities. Furthermore, a modular adaptation policy language allows engineers to consider adaptation concerns separately and then compose them in the context of a specific system.

Rainbow makes adaptation decisions using two kinds of models. The architecture model reflects abstract, run-time states of the target system itself. The environment model provides contextual information about the system, including its executing environment and computational resources.

The core Rainbow framework is implemented in Java. Elements below the translation layer may be implemented in a language or script of choice, but must conform to the framework's probe and effector communication protocols. At run time, a *Rainbow Master* instantiates the Architectural-Layer elements shown in Figure 1. A *Rainbow Delegate* is deployed on each computing node of the target system to manage probes, gauges, and effectors on that node. An event bus coordinates communication between Master and Delegates.

We now describe our benchmarking system, Znn.com, that we used to evaluate Rainbow.

3.2 The Znn.com System

The typical infrastructure for a news website like cnn.com and rockymountainnews.com has a three-tier architecture consisting of a set of application servers that serve contents from backend databases to clients via frontend presentation logic. The Znn.com system imitates such a setup. Architecturally, it is a web-based client-server system that satisfies an N-tier style, as illustrated in Figure 2. Znn.com uses a load balancer to balance requests across a pool of replicated servers, the size of which can be manually adjusted to balance server utilization against service response time. A set of client processes makes stateless content requests from one of the servers, the servers deliver static files (e.g., images and videos), as well as dynamic content (e.g., news populated from periodically-updated sources).

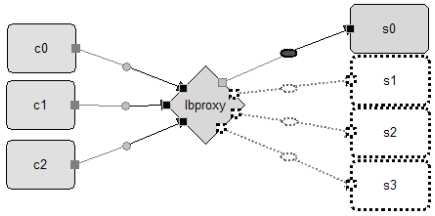

Figure 2. The Znn.com system architecture

Typical of news provider concerns, our quality objective for Znn.com is to serve news content to its customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. From time to time, due to highly popular events, Znn.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, we opt to serve minimal textual contents during such peak times in lieu of providing zero service to the customers. In short, we identify three quality objectives for the self-adaptation of the Znn.com system: (A) performance, (B) cost, and (C) content fidelity.

Performance analysis suggests we monitor the request-response time, server load, and connection bandwidth of the system. Cost analysis identifies the number of active servers as the primary contributor to cost; hence we monitor the server count. For content fidelity, we characterize different levels of content ranging from full multimedia to static text, assigning three levels (high, medium, and low). The major elements of the N-tier-client-server architectural style for Znn.com include:

- Types: ClientT, ServerT, ProxyT, HttpConnT
- Properties: ClientT.experRespTime, ServerT.cost / load / fidelity, HttpConnT.bandwidth
- Operators: ServerT.activate() / .deactivate() / .setFidelity(level : int)

The ServerT.activate() operator activates a ServerT instance, while the deactivate() operator deactivates it. The ServerT.setFidelity(level : int) operator sets the server content fidelity to the level identified by the input parameter. Using these operators, we specified two pairs of tactics with opposing effects. One pair enlists

(1) or discharges (2) servers while the other pair raises (3) or lowers (4) the server content fidelity. In effect, these tactics allow the service level of the Znn.com system to be stratified into gradients that trade off the various objectives. The following example illustrates how these tactics might interact:

When response time is high, objective A (above) suggests that Znn.com should increment its server pool size (using tactic 1 above) if it is within budget; otherwise, Znn.com should switch the servers to textual model (using 4). When the response time is low, objective C suggests that Znn.com should decrement its server pool size (using 2) if it is near budget limit; objective B suggests that Znn.com should switch the servers to multimedia mode (using 3) if they are not already in that mode. When the response time is in the normal range, objective B suggests that Znn.com should switch the servers to multimedia mode if they are currently textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

We have further defined four adaptation strategies from these tactics, with juxtapositions that allow system adaptation to balance the overall objectives:

- SimpleReduceResponseTime: When any client experiences a response time above threshold, lower content fidelity one step, then lower fidelity again if response time is still above threshold.
- SmarterReduceResponseTime: Let *n* be the count of clients experiencing above-normal request-response time; if *n* exceeds a tolerable percentage of total, enlist a server, then enlist another server, then lower the fidelity one step, then repeat the last sequence twice until successful.
- ReduceOverallCost: When the total server cost exceeds a threshold value, discharge up to four servers, one at a time, until the cost is reduced below threshold.
- ImproveOverallFidelity: When the average content fidelity of the servers drops below a threshold value, raise the fidelity level for all servers, up to twice, until average fidelity rises above threshold.

To summarize, the quality dimensions, architectural element types and properties, and adaptation operators, tactics, and strategies together comprise the artifacts to customize Rainbow to Znn.com. We now turn to the evaluation of Rainbow using Znn.com.

4. Evaluation with Znn.com

The objective of the Rainbow research is to enable software engineers to build self-adaptive systems cost-effectively. Hence, the success of Rainbow depends directly upon how costly it is to use and how well the resulting system self-adapts. Consequently, evaluating the effectiveness of Rainbow requires showing that:

1. The Rainbow-customized target system self-adapts to provide improved overall utility
2. Self-adaptation incurs low run-time resource overhead, and
3. The effort required to tailor Rainbow to the target system is significantly lower than developing the self-adaptive capabilities from scratch

We evaluated Rainbow on five example systems, the details of which are found in Cheng's dissertation [4]. In this section, we present the Znn.com evaluation data to show how it satisfies the three criteria.

4.1 Show that Znn.com Is Self-Adaptive

To evaluate the effectiveness of Rainbow's self-adaptation on a news website, we built the Znn.com system using open-source, commercial software. We "Slashdot-effect" experiment on Znn.com.

The setup consisted of a pool of four typical Intel (~1 GHz) machines, each running a Debian-flavor Linux operating system, configured with an instance of the Apache webserver. A fifth machine ran a load balancer to forward incoming requests in a round-robin fashion to any active server among the four. Two additional machines were set up to act as the clients, using Apache JMeter, a Java application for testing web applications and measuring their performance, to simulate request loads from multiple clients.

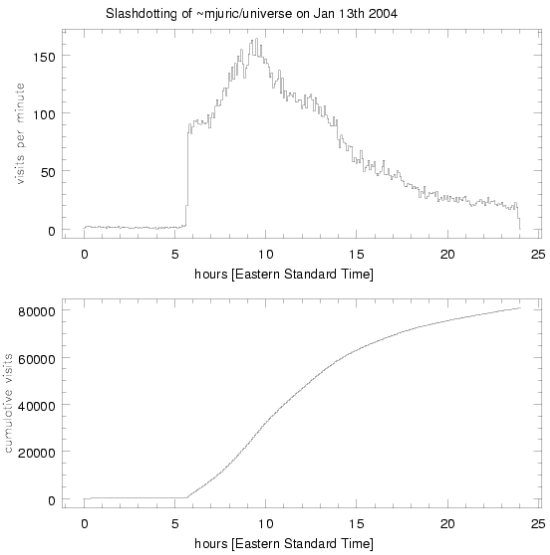

Figure 3. Graph of actual, peak-day traffic of a site experiencing *Slashdot effect*

To perform the experiment, we designed a workload that is characteristic of a *Slashdot effect* visitor traffic profile, based on a sample collected by ~mjuric and shown in Figure 3 [15]. Due to the resource-demanding nature of the Slashdot effect, an adaptation that does not quickly offset the sudden rise in demand for resources would not be effective. Therefore, the initial sharp rise in traffic entails the critical duration of interest for our experiment purposes. For measurement purposes, we choose to observe a sustained duration after the initial rise to make sure that any effective adaptations remain effective for a reasonable amount of time. In lieu of 12 to 18 hours of actual traffic, we patterned our traffic profile after the ~mjuric profile, scaled down to one hour (12:1) but kept at a similarly high visit rate:

1. 1 minute of low activity, 6 unique visits/min
2. 5 minutes of sharp rise in requests, ramping up to 600 visits/min (+120 visits/min/min)
3. 18 minutes of peak in requests, sustained at 600 visits/min
4. 36 minutes of linear decrease, ramp down to 60 visits/min (-15 visits/min/min)

We constructed this workload in JMeter, using a Gaussian random timer between requests. We then deployed this workload on two JMeter instances to generate the news reader traffic for our Znn.com example. Finally, we devised the following trial types to assess Rainbow's effectiveness at adapting Znn.com. For each trial type, we performed five runs to smooth stochastic anomalies and to yield consistent outcomes:

1. Control runs without Rainbow adaptation – to establish baseline and comparison envelopes
2. Experimental runs with Rainbow adaptation

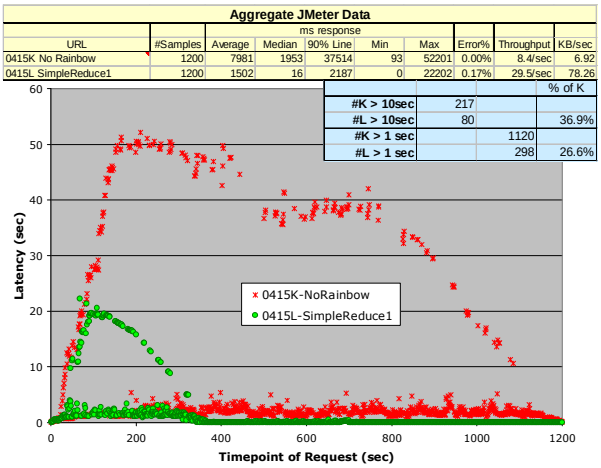

Figure 4. Znn.com experiment data

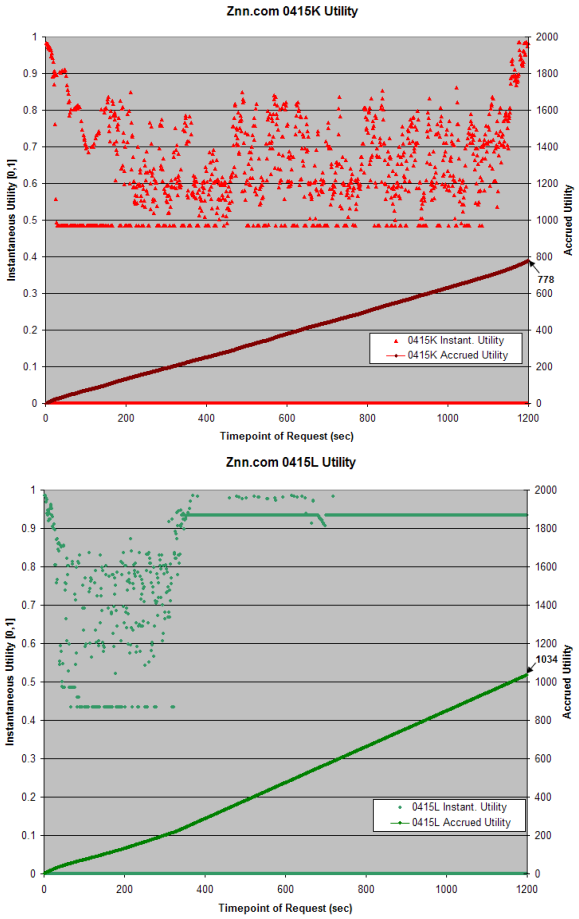

Figure 5. Instantaneous and accrued utility

For every run, we collected statistics on the total number of samples, response latencies, request throughput, and any errors. We also tracked the corresponding cost and content fidelity values to compute accrued utility and provide a complete picture of the tradeoff space as defined by the overall objectives.

Figure 4 shows a graph for two experiment runs, control (red) versus adaptation using one simple strategy (green). For each run, the JMeter data table shows the total count of request samples, the statistics of request-response time, and the net throughput. The graph plots the latency per request, and the small table summarizes how many requests yielded latencies above 10 seconds and 1 second. The data indicated that Znn.com with Rainbow adaptation, in contrast to Znn.com without, yielded far lower latencies (902 of 1200, or 75%, requests served within 1 second vs. 80 of 1200, or 7%) and better throughput (3.5×). Therefore, Rainbow was effective at keeping user latencies low and managing the Slashdot effect.

Figure 5 shows graphs of instantaneous utility (IU) and accrued utility (AU) for the two runs, which indicate how well the system satisfied overall user objectives. Each IU value is computed as the weighted sum of utilities over the response time (0.50), content fidelity (0.25), and cost (0.25) dimensions. At about 360s, after an initial period of chaos, the IUs in Znn.com with adaptation remain high compared to Znn.com without adaptation. The final AU values show a 33% improvement (1034 vs. 778) in Znn.com with adaptation.

4.2 Show that Resource Overhead Is Low

Znn.com allows us to estimate overheads of adaptation mechanisms by providing a baseline with which to compare the system before and after adding adaptation. To get a baseline measurement, we measured CPU and memory usages solely with Znn.com components. We then compared them with the same measurements take with the Rainbow Delegates deployed.

Resource overhead incurred by the Delegate was minimal, consuming less than 2% of CPU (but occasionally sustained at ~5-10%) with ~2MB of memory footprint. This overhead would be further reducible with optimization in implementation. Nevertheless, on a computing node with highly constrained memory and CPU resources, the adaptation engineer might choose to deploy only probes on the node and configure the probes to report via neighboring Delegate nodes.

4.3 Show Low Customization Effort

While obtaining empirical metrics for any human-based engineering activity is often a complex process (involving many test subjects, development conditions, etc.), it is often possible to obtain "order of magnitude" comparisons with less effort. Znn.com allows this kind of evaluation by facilitating a task-based estimation of efforts to engineer or retrofit adaptations.

To evaluate how long it took to customize Rainbow for Znn.com, we tracked our customization activities in detail [4], summarized here. The customization effort, including architecture modeling (using Acme), adaptation scripting (in Stitch), and development and testing of probes, gauges, and effectors (in Perl, shell script, and Java), accounted for a total of 93 hours, or approximately 2 1/3 work weeks. Of this, 13 hrs (14%) were used to describe the model, 49 hrs (53%) to develop probes and gauges, 7 hrs (8%) to develop effectors, 21 hrs (23%) to compose adaptation scripts, and 3 hrs (3%) to put together the customization files. Note that while the majority of the effort was spent developing monitoring capabilities, the resulting probes and gauges are reusable artifacts, so less effort would be required as more are developed. Furthermore, the order of magnitude of effort has greater significance than the actual durations: most activities required on the order of minutes to a couple hours, not days, while incremental changes required on the order of tens of minutes, not hours.

To compare this effort with building in adaptation capabilities from scratch, we decomposed the self-adaptation engineering process into four coarse-grained tasks – three development tasks and one evolution tasks – and estimated the time to complete each task: *domain analysis*, *model capture*, *design and implementation*, and *updates and modifications*. As evidence, we gathered development-activity data from our case examples and performed exploratory analysis of hypothetical custom-solution scenarios. To prevent skewing the comparison in favor of Rainbow, we made estimates that favored custom-solution wherever possible.

For the custom-solution efforts, we assumed that domain analysis done by a domain expert required the same amount of time as with Rainbow, that model capture generally required zero time (i.e., no model capture), that design and implementation by an expert software developer team required a minimum of one man-month on top of the Rainbow-based time for developing the monitoring and effecting capabilities, and that each update and modification required one-quarter the time required for design and implementation because of buried and dispersed adaptation logic.

We can reasonably assume that, Rainbow or not, similar probing and effecting mechanisms would have to be implemented into the target system, though perhaps with a simpler reactive mechanism created in place of Rainbow's architecture model and Adaptation Manager. Also, we can reasonably expect that similar deployment and roundtrip debug efforts would be required. Additional effort would be necessary for the basic adaptation plumbing. Based on the Rainbow framework development experience, it took one expert Java developer over 2.5 man-months to design and implement the communication infrastructure and plumbing for the probes, gauges, and effectors. Even if we assume simpler requirements in the case of building from scratch, incurring only half the time of developing Rainbow, it would still yield a total effort of more than one man-month to add adaptation capabilities into the target system. (In doing so, one would also lose Rainbow's engineering advantages of architecture-level modeling and analysis, separation of adaptation concerns from system functionality, and flexibility to evolve self-adaptation capabilities.)

We treat the Znn.com system as an average-case scenario due to its correspondence with typical N-Tier IT systems; for the Rainbow-based data, we estimated about two weeks of domain analysis. While we do not have concrete data for a worst-case scenario, we estimated worst-case Rainbow-based development time to be an order-of-magnitude longer than the average case.

In the worst-case scenario, a Rainbow instantiation has no reusable style, gauges, probes, and effectors from prior efforts. The adaptation engineer must construct many elements from scratch, so the primary advantage of Rainbow comes from the reusable framework. Thus, Rainbow-based initial development does no better in the worst case than a custom solution. According to our coarse-grained task estimation of efforts, excluding the worst case, Rainbow yielded effort savings of 2 to 5 times over custom solution for initial development of self-adaptation capabilities. Thereafter, Rainbow achieves additional savings of 6 to 192 times, or up to two orders-of-magnitude, over custom solution when evolving self-adaptation capabilities. The upfront effort of engineering the generic Rainbow framework accounts for the time savings over custom solution.

In summary, Rainbow makes Znn.com self-adaptive, incurs less than 5% resource overhead on average, and requires low customization effort on the order of days to weeks. We now make the case for using Znn.com as a common benchmark system.

5. What Makes a Good Benchmark?

While evaluating Rainbow, we realized that the Znn.com system has the potential to serve as a useful example for researchers in the community to compare techniques. In this section, we enumerate a set of general requirements for a benchmark system, present Znn.com as a candidate, and pose a number of benchmark issues for discussion.

5.1 Benchmark Requirements

When we consider computing benchmarks in areas such as CPU [26], databases [10], and algorithms [11], we observe that a benchmark system, in general, should satisfy the following requirements:

- *Relevance* to real-world problems
- *Accessibility* of system components and codebase that can be tested in a *standalone* environment
- *Capability* to *observe* and *change* the system, by mechanisms that range from changing component parameters to altering overall system configuration
- *Metrics* for comparison, for example, CPUs have transistor count and MIPs, algorithms have duration in seconds over input size, and databases have throughput transactions per second
- Ease of *extracting* performance data from the system and environment for conducting evaluation

Specific to the area of self-adaptive systems, we see a number of additional requirements on the benchmark system to facilitate comparison and contrast of essential features of self-adaptation:

- *Versatility* to apply to a wide range of self-adaptive approaches and application domains
- Allow changing the system *dynamically*
- Support multiple quality *dimensions* (trade-off), e.g., availability, performance, reliability, security
- Allow multiple adaptive *operations*, e.g., enabling or disabling servers, altering connections, tuning component parameters, quiescing processes
- Provide multiple *alternatives* to achieve a single operation, e.g., disabling a server by killing a process or powering down the machine, changing content fidelity by swapping configuration files or via an Apache plug-in
- Provide multiple paths of system *configuration* to achieve the same goal, e.g., increasing overall throughput by adding servers followed by lowering fidelity, or in the reverse order, or by some completely different sequence of operations.

Therefore, a useful benchmark system is relevant, accessible, dynamically observable and changeable, and versatile; supports alternative adaptive operations and multiple configurations; facilitates quality trade-offs; and can be compared via a common metric.

5.2 Znn.com a Candidate Benchmark System?

While any system that meets the requirements laid out above may be a candidate benchmark system, Znn.com is especially fitting because it is constructed from openly accessible software components, and it has potentially many controlled variations.

Znn.com has relevance. Assessed abstractly, the Znn.com system contains important features of a real-world problem: It mimics the infrastructure of actual news-provider services. The *Slashdot effect* is a resource allocation problem that occurs in real systems, that lacks a perfect solution and also has impact on multiple quality dimensions, making it amenable to self-adaptive techniques, especially those managing multiple objectives.

Znn.com is built from open-source software that is available for the major platforms, including Linux, Mac, and Windows: the Apache Webserver, PHP, and

the MySQL database server. The supporting tool suites are also widely accessible, including Perl and JMeter.

Out of the box, the software components do not directly enable dynamic observation and change. For those purposes, we constructed a suite of *probes* and *effectors*. We chose to probe some common system properties – CPU load, disk I/O rate, and available bandwidth between nodes – and a few domain-specific properties – Apache status and server content fidelity. The probes to detect CPU *load*, *bandwidth*, and content *fidelity* were implemented as Perl scripts. The Apache status probe was derived from the C program *apacheTop*. Disk I/O was a ported *iostat* program.

To effect changes in Znn.com, we developed a set of Perl scripts to start or stop a server process (turnServer), to alter the webserver content fidelity (changeFidelity), and one to randomly reject client requests (setRandomReject). To alter content fidelity, we defined three sets of webpages with high, medium, and low fidelity content, and wrote corresponding httpd configuration files. Altering the fidelity consisted of swapping out httpd files and gracefully restarting the active Apache processes. Another, more involved approach would be to develop a plug-in module that allows setting different *levels* of Apache service.

Note that while these probes and effectors were appropriate for our quality goals, not only were they reusable artifacts in Rainbow, they could also be reused in a variety of contexts where Apache is used or where CPU load, disk I/O, and bandwidth information is required. Following a similar scheme, it should be easy to develop other probes and effectors depending on the specific domains of adaptive systems and the qualities being demonstrated.

The ability to extract performance data from the target system and environment for conducting evaluation depends partly on the specific self-adaptation framework and partly on the testing tool. For Rainbow, we developed gauges such as the ApacheTop Gauge, Latency Gauge, and End-to-End Response-Time Gauge to read probe data and update the architecture model. The Adaptation Manager then dumps relevant data (e.g., adaptation duration or memory used) to a log file. We also relied on the testing tool, JMeter, to obtain the necessary measurements, including response time and throughput. JMeter also provides features to visualize and export data to facilitate analysis.

We targeted three quality dimensions for evaluating Znn.com, as shown in Table 1. However, Znn.com is capable of catering to a variety of quality dimensions, including different performance concerns, availability, reliability, security, and even data privacy. In addition, Znn.com allows interesting trade-off considerations across quality dimensions, for example, sacrificing just the cost to reduce client-experienced response time, or just the content fidelity, or both cost and fidelity.

Table 1. Quality dimensions and the corresponding probes and effectors

Dimension	Probe	Effector
Client-experienced response time	CPU load, bandwidth, diskIO[1]	turnServer, changeFidelity, setRandomReject[1]
Provision cost	apacheTop	turnServer
Content fidelity	fidelity	changeFidelity

As our example demonstrated, the Znn.com example is rich enough to provide for variety at three levels: multiple adaptive operations, multiple alternatives for each operation, and multiple paths of configuration. In short, Znn.com meets eight of the ten requirements; it does not meet the versatility and metric requirements. Regarding the latter, while no single basis of measurement seems appropriate for all self-adaptive systems, we believe that we can apply utility theory to enable comparison across seemingly different and incompatible dimensions of self-adaptation. In this way, we may be able to arrive at a common basis for comparison.

The key idea to using utility theory is to explicitly enumerate the quality dimensions for which one is evaluating the self-adaptation technique of choice; then, indicate the weight given to each dimension. The weighted sum of scores across the dimensions yields a single value, which we shall term the Self-Adaptation Fitness Unit (SAFU). In benchmarking an approach, one may therefore opt to account for dimensions relevant to the approach. In the case of Rainbow, for example, three important evaluation criteria are (1) meeting quality of service goals, (2) resource overhead, and (3) adaptation engineering effort. We factor in different weights for these criteria, as shown in Table 2.

Table 2. SAFU for Rainbow on Znn.com

Criteria	Value		Weight	
Quality dimensions	86 (1034/1200 AU)		40%	
- Response time	994		50%	
- Content fidelity	1021		25%	
- Provision cost	1128		25%	
Resource overhead	90-95 (5-10% overhead)		30%	
Engineering effort	30-45 (days-weeks)		30%	
Self-Adaptation Fitness Unit (SAFU):			**70-76**	

[1] Probe and effector that were developed but, in the end, we did not need in our evaluation.

An interesting issue is how to quantify each criterion. We scored the resource overhead as a percentage of usable resources, or 100 minus an average overhead of 5-10%, or 90-95. We scored engineering effort by mapping time orders-of-magnitude (minute, hour, day, week, month, year) in seconds (60, 3600, 86400, …) to a logarithmic scale, normalized to 100 (100, 69, 45, 30, 19, 0); an effort of days-weeks scores 30-45. These component scores yield a SAFU in the range of 70-76:

$$70 = 86\times40\% + 90\times30\% + 30\times30\%$$
$$76 = 86\times40\% + 95\times30\% + 45\times30\%$$

A SAFU of 100 indicates the perfect achievement of all adaptation goals, which should be unreachable. The comparison between any two self-adaptation techniques comprises: (a) their relative SAFU and (b) how they contrast in dimensions and weights.

5.3 Discussion

We introduced the Znn.com system as a candidate benchmarking example for self-adaptive systems. While its broad utility as such is yet to be shown, our intent in offering it here is partly to spur community discussion on what would make good benchmarks in this domain. In particular, the following issues are worth discussing:

Engineering cost: In [19] McCann discussed nine evaluation metrics for self-adaptive systems. While we evaluate Rainbow here with respect to two of these, we believe that many of the others could be captured with this example. For example, stabilization and adaptivity would be relatively trivial to measure. Furthermore, we believe another important metric is the cost to engineer adaptation, especially when being applied to legacy systems. We measured effort by tracking the amount of time spent engineering each of the components of Rainbow for Znn.com, and used estimates and interviews to compare this to engineering self-adaptation without Rainbow. However, this raises the issue of how engineering effort can be compared.

SAFU as a common quantitative metric: While it remains to be seen whether the SAFU will prove useful as a common metric for comparing different self-adaptation techniques, one might argue that wide variance in dimensions and weights renders comparison meaningless. However, at the very least, it provides a conceptual (utility-based) framework for deriving shared *profiles* of SAFU in the future. Each profile would define a common set of dimensions and weights that is applicable to a particular class of self-adaptation techniques. Ours represents a profile instance in the class of architecture-based self-adaptation. A profile in a different class, such as resource-constrained mobile-computing self-adaptive systems, might have completely different dimensions, e.g., mobility, component robustness, sensor heterogeneity. But what constitutes a sufficient set of evaluation points? Are there other measurements that can be used generally for fair comparison? For example, should one count resource overheads during no-adaptation periods?

Versatility: While Znn.com is mainly useful for evaluating web-based information-system adaptation, there are many other domains to which self-adaptation has been applied – for example, mobile and embedded systems. In fact, these domains may already have existing benchmarks. McCann suggested adding autonomic benchmarking to existing benchmarks in other domains, rather than developing one for autonomous computing [19]. We agree that this might be a way to address the versatility requirement: one would start with an existing benchmark for the target domain, then add relevant *autonomic benchmarks*, and apply SAFU to derive a quantitative score for comparison. Here, the interesting issues to discuss regard what autonomic benchmarks would look like for specific domains and how to quantify them for comparison. For example, there might be specific benchmarks for mobility, multiple environmental contexts (ubicomp), dynamic components (genetic programming), and different self-* properties (self-organizing systems). Again, SAFU profiles might help in elucidating these.

6. Conclusion and future work

In this paper, we reported on our evaluation of the Rainbow self-adaptation approach using Znn.com. We presented a list of requirements for a benchmark environment to facilitate meaningful comparison with other self-adaptive techniques, and reflected upon how well Znn.com meets those benchmark requirements.

In addition to reporting on the evaluation of Rainbow and being instructive on how to apply an architecture-based self-adaptive system to Znn.com, we hope that this paper will generate discussion in the community about what is needed for comparing different self-adaptation approaches. Finally, we make the benchmark tool suite available for general community use at this URL: *http://rainbow.self-adapt.org/benchmark* .

In future work, we would generalize the instance into a benchmark "framework" that facilitates plug-in of any components catering to specific self-adaptation agenda. We would also create web resources to collect and disseminate experience with its use.

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 grants DAAD19-02-1-0389 to Carnegie Mellon University's CyLab and DAAD19-01-1-0485, and the NSF under grants CNS-0205266, 0615305, 0834701, and IIS-0534656. 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 the funders, the US government, or any other entity.

References

[1] G.D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. ACM Software Engineering Notes, 18(5):9-20, 1993.

[2] T.V. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In EWSA, LNCS 3527:1-17, Springer, June 13-14, 2005.

[3] J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In WOSS '04: Proc. of the 1st ACM SIGSOFT Workshop on Self-managed Systems, pp. 28-33, ACM, New York, NY, 2004.

[4] S.-W. Cheng. Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. Ph.D. Dissertation, TR CMU-ISR-08-113, Carnegie Mellon University School of Computer Science, May 2008.

[5] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. (eds.) Proc. of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), pp. 21-26, New York, NY, USA, ACM Press, November 18-19, 2002.

[6] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli. A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1(2), 223-259,2006.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Computer 37(10), 2004.

[8] I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architectures for distributed systems. In Proc. 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), pp. 33-38, ACM Press, New York, NY, USA, 2002.

[9] M.M. Gorlick and R.R. Razouk. Using Weaves for software construction and analysis. In Proc. of the 13th International Conf. of Software Engineering, pp. 23-34, Los Alamitos, CA, USA, IEEE Comp. Society Press, May 1991.

[10] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[11] G.T. Heineman, G. Pollice, and S. Selkow. Algorithms in a Nutshell, O'Reilly, 2009.

[12] M.C. Huebscher, J.A. McCann. A Survey of Autonomic Computing – Degrees, Models, and Applications. ACM Computing Surveys 40(3), 7, 1-28, 2008.

[13] IBM. An architectural blueprint for autonomic computing, 2004.

[14] IBM developerWorks. Autonomic computing toolkit. ibm.com/developerworks/autonomic/overview.html, 2008.

[15] M. Juric. Slashdotting of mjuric/universe. www.astro.princeton.edu/~mjuric/universe/slashdotting/, January 13–15, 2004.

[16] J.O. Kephart and D.M. Chess. The vision of autonomic computing. IEEE Computer, 36, 1, Jan 2003.

[17] J. Magee and J. Kramer. Dynamic structure in software architectures. In SIGSOFT '96: Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 3-14, New York, NY, USA, 1996. ACM.

[18] V. Markl, G.M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. IBM Systems Journal, 42(1):98–106, 2003.

[19] J.A. McCann and M.C. Huebscher. Evaluation Issues in Autonomic Computing. In Proc. International Workshop on Agents and Autonomic Computing and Grid Enabled Virtual Organizations (AAC-GEVO'2004), LNCS 3252, 2004.

[20] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R.M. Greenwood. An active architecture approach to dynamic systems co-evolution. In ECSA, LNCS 4758:2-10. Springer, September 24-26, 2007.

[21] A. Mukhija and M. Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops – W7: EC (ICDCSW'04), pp. 368-374, IEEE Computer Society, Washington, DC, 2004.

[22] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Syst., 14(3):54-62, May-June '99.

[23] L.W. Russell, S.P. Morgan, and E.G. Chron. Clockwork: A new movement in autonomic systems. IBM Systems Journal, 42(1):77–84, 2003.

[24] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu. A Real-Time Adaptive Control of Autonomic Computing Environments. In Proc. 4th Int'l Information and Telecommunication Technologies Symposium (U2TS'2006), 2006.

[25] A. Sztajnberg and O. Loques. Describing and deploying self-adaptive applications. In Proc. 1st Latin American Autonomic Computing Symposium, July 14-20, 2006.

[26] R.P. Weicker, An Overview of Common Benchmarks, Computer, 23(12): 65-75, December, 1990.

[27] A.L. Wolf, D. Heimbigner, A. Carzaniga, K.M. Anderson, and N. Ryan. Achieving survivability of complex and dynamic systems with the Willow framework. In Proc. of the Working Conf. on Complex and Dynamic Systems Architecture, December 12-14, 2001.