To appear in Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee (eds). Software Engineering for Self-Adaptive Systems. LNCS 2008.

Improving Architecture-Based Self-Adaptation Through Resource Prediction

Shang-Wen Cheng, Vahe V Poladian, David Garlan, Bradley Schmerl

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

Abstract. An increasingly important concern for modern systems design is how best to incorporate self-adaptation into systems so as to improve their ability to dynamically respond to faults, resource variation, and changing user needs. One promising approach is to use architectural models as a basis for monitoring, problem detection, and repair selection. While this approach has been shown to yield positive results, current systems use a reactive approach: they respond to problems only when they occur. In this paper we argue that self-adaptation can be improved by adopting an anticipatory approach in which predictions are used to inform adaptation strategies. We show how such an approach can be incorporated into an architecture-based adaptation framework and demonstrate the benefits of the approach.

Keywords: self-adaptation, resource prediction, autonomic computing, software architecture

1 Introduction

As computing systems become more and more integral to our daily activities, it becomes increasingly important for those systems to provide reliable and uninterrupted service, even in the presence of system faults, changing resources and loads, and different user needs. In the past this capability has largely been provided through human oversight. As a result, the cost of managing such systems has grown to 70-90% of the total cost of system ownership [8], while the burden of managing the many aspects of computing has surpassed the capacity of human attention [24].

In response there has been considerable recent interest in supporting automated system self-adaptation, whereby the system takes increasing responsibility for dynamically detecting problems and repairing itself. Most systems that support self-adaptation adopt a control systems perspective: a system is monitored and the resulting observations are used to determine system health, and the system is adapted to fix any existing problems.

One particularly promising form of this approach is to use architectural models of a system as the basis for problem detection, diagnosis, and repair. Architecture-based self-adaptation has had considerable success in providing adaptation support for legacy systems and in providing flexibility for tailoring adaptation to business needs [1,6,10,11,12,13,19,25].

One outstanding problem with such systems is that they are strictly reactive: they respond to the current environment and system state, invoking adaptation strategies if and only if an immediate problem arises. The goal of a reactive approach is to select an adaptation that optimizes the instantaneous utility of the system at that time. However, from a global perspective, several instantaneously optimal decisions may be sub-optimal when considered together. For example, if we adapt a web system reactively to a short, temporary spike in bandwidth by reducing the fidelity of the content, this may be sub-optimal in hindsight because a short delay may be less offensive to the client than low fidelity.

In this paper we argue that self-adaptation can be dramatically improved if we use *future predictions* of the environment, and specifically its resources, to make better choices about whether and how to adapt a system. In other work [21], we have developed a resource prediction framework that provides predictions on resource availability from a variety of prediction models, in the context of continually adapting ubiquitous computing. We can use this framework to provide predictive information to help architecture-based self-adaptation. In particular, we observe that prediction offers four kinds of improvement to the existing self-adaptation approach:

1. Prediction prevents unnecessary self-adaptation.
2. Prediction reduces disruption from incremental adaptation, for example, enlisting servers 4 at once rather than one at a time.
3. Prediction enables pre-adaptation to seasonal behavior.
4. Prediction improves overall choice of adaptation.

At first glance, it seems obvious that using predicted information will improve self-adaptation – if you know it is going to rain, don’t turn on the sprinklers. But, making choices about when and how to consider this predicted information is crucially important. Accordingly, the contributions of this chapter are:

1. A framework for generic use of predictive information. The framework is agnostic to methods used for deriving predictions;
2. Flexibility in using predictions for self-adaptation. Our framework has several points of integration where predictions can be useful; and
3. Some rules-of-thumb for how to incorporate predictive information into a self-adaptive framework.

In the remainder of this chapter we describe our resource prediction framework and show how it achieves the improvements listed above. In Section 2, we describe the overall framework of our architecture-based self-adaptation approach and identify core challenges of incorporating prediction. We then introduce the anticipatory model for adaptation in Section 3. In Section 4 we present initial results of applying an anticipatory model to adaptation and describe future applications. In Section 5 we describe related work on architecture-based self-adaptation and prediction. In the final section, we conclude with a brief discussion of additional ways in which prediction could be used to improve architecture-based self-adaptation.

2 Framework for Architecture-Based Self-Adaptation

In this section we provide a high-level overview of our self adaptation framework, illustrate its use with an example, and discuss opportunities for enhancement via resource prediction. In particular, making use of prediction requires addressing a few challenges: What kinds of predictive information are useful? What can be predicted? How would it be used? This section addresses the first question of requirements for prediction. In the next two sections we address the questions of what and how.

To illustrate our approach, consider an example news service, Znn.com, inspired by real sites like cnn.com and RockyMountainNews.com, which serves multimedia news content to its customers. Architecturally, Znn.com is a web-based client-server system that conforms to an N-tier style. As illustrated in Fig. 1, Znn.com uses a load balancer (LB) to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization against service response time. A set of client processes (represented by the C component) makes stateless content requests to the servers. Let us assume we can monitor the system for information such as server load and the bandwidth of server-client connections. Assume further that we can modify the system, for instance, to add more servers to the pool or to change the fidelity of the content. We want to add self-adaptation capabilities that will take advantage of the monitored system and adapt the system to fulfill Znn.com objectives.

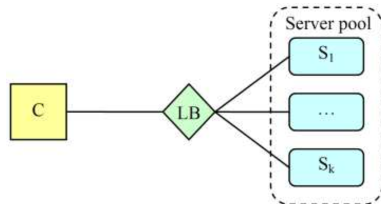

Fig. 1. Architecture model of Znn.com

The business objectives at Znn.com are to serve news content to its customers with reasonable response, while keeping the cost of the server pool within its 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 unacceptable latencies, Znn.com opts to serve minimalist textual content during such peak times in lieu of providing its customers zero service. Assume that two actions are possible to adapt the system: adjust the server pool size (enlist or remove) or switch content mode (multimedia or textual). While seemingly simple, an adaptation decision requires a tradeoff between the multiple objectives.

2.1 Overview of the Rainbow Framework

Our architecture-based self-adaptive approach is embodied in an engineering framework, called Rainbow, which provides mechanisms to monitor a target system and its executing environment, reflect observations in an architecture model, detect opportunities for improvements, select a course of action, and effect changes. By leveraging the notion of *architectural style* to exploit commonality between systems, the framework provides general and reusable infrastructures with well-defined customization points to cater to a wide range of systems. It also provides a useful set of abstractions

to focus engineers on adaptation concerns, facilitating the systematic customization of Rainbow to particular systems. Details can be found in [3,4].

The Rainbow framework (Fig. 2) uses a component-and-connector architecture model of the target system to monitor and reason about appropriate strategies for adapting the system. Monitoring mechanisms—*probes* and *gauges*—observe the running *target system*. Observations are reported to update properties of the 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 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 and choose an appropriate adaptation strategy. The *Strategy Executor* then executes the strategy on the running system via system-level *effectors*.

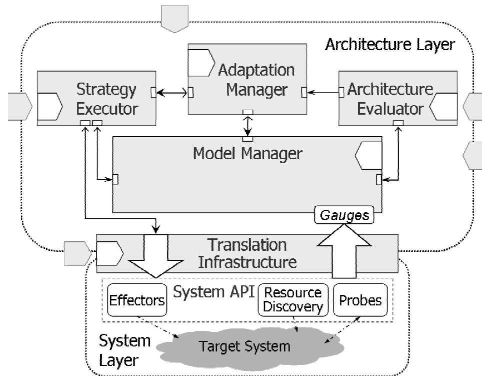

Fig. 2. The Rainbow Framework

To apply Rainbow to the Znn.com example, we use probes and gauges to monitor response time and server load, reflecting those as properties in the architecture model. The architecture evaluator triggers adaption when any client experiences request-response latencies above some threshold. The Adaptation manager determines whether to activate more servers or decrease content fidelity, as specified in a repair script. The strategy executor effects the change in Znn.com using provided hooks.

When the system comes under high load, Rainbow may opt to increase the server pool size until a cost-determined maximum is reached, at which point Rainbow would switch the servers to serve textual content. If the system load drops, Rainbow may switch the servers back to multimedia mode to make customers happy, in combination with reducing the pool size to reduce operating cost. In general, the adaptation decision is determined by both the business objectives and observations of system conditions, including average response time, server load, and available bandwidth.

2.2 Elements of Rainbow

The Rainbow framework uses models of the architecture and environment to make adaptation decisions. A component-and-connector (C&C) architecture model reflects abstract, runtime states of a target system, including what entities are present and how they communicate [5]. An environment model provides contextual information about the system, including its executing environment and the resources used. For example, when additional servers are needed, the environment model indicates what spare servers are available. When a better connection is required, the environment model contains information about the available bandwidth of other communication paths.

In order to get information out of the target system into an abstract model for management, and then to push changes back into the system, we need mechanisms that hook into the target system and understand what is represented in the model. Gauges process system-specific information from Probes to populate architectural properties. Associated with architectural operators in the Rainbow Architecture Layer, effectors carry out change operations on the target system via mechanisms that range in complexity from a system-call, to a script, to an elaborate workflow.

The Architecture Evaluator evaluates model conformance against architectural constraints, which are specified using first-order predicate logic to identify problems in the system. When triggered by the Architecture Evaluator, the Adaptation Manager uses information about the state of the system, embodied in the architecture, the business quality-of-service concerns and utility functions to decide which remedial strategy to execute. A strategy is chosen from a set of specified strategies that have been engineered for the system and/or domain. A strategy specifies conditions and contexts in which it applies, and captures a pattern of adaptation steps.

Business quality-of-service concerns for the target system (e.g., system reliability, service availability, or performance) are represented as quality dimensions. A quality dimension provides a notion of utility, or happiness, for particular values of a quality attribute. Each adaptation action has a specified impact in cost or benefit on each dimension. By tallying the *cost-benefit attributes* over the actions in a strategy, an expected aggregate impact can be computed for each strategy. A strategy can then be scored using utility preferences specified for the quality dimensions. The Adaptation Manager then selects the highest-scoring strategy.

Utility preferences define the relative importance between the quality dimensions. Specifically, we use a von Neumann-Morgenstern utility function $u_d : X_d \rightarrow \Re$ that assigns a real number to each quality dimension d, normalized to the range [0,1]. Across multiple dimensions, we attribute a percentage weight to each dimension to account for its relative importance compared to other dimensions. These weights form the utility preferences. The overall utility is then given by the utility preference function, $U = \Sigma w_d u_d$. An example utility preference with three objectives, u_1, u_2, u_3, of decreasing importance might be quantified as [w_1:0.6, w_2:0.3, w_3:0.1].

The utility preference function gives us a way to compute the *instantaneous utility* of the target system given its current conditions, as well as the *accrued utility* of the target system over time. If we assume coverage of system conditions, accrued utility provides a measure of optimality of the target system, giving us a way to compare the relative optimality of a system under different combinations of conditions.

2.3 Opportunities for Improving Self-Adaptation

To date, Rainbow's adaptation has been *reactive* in nature. Reactive adaptation has the advantage of requiring only a small set of recent system conditions to choose an adaptation, allowing for timely decisions. However, reactive adaptation has a number of well-known disadvantages. First, following the decision to perform an adaptation, time is needed to carry out and propagate the necessary changes on the target system. At times, the conditions that trigger an adaptation may be more short-lived than the

duration for propagating the adaptation changes, resulting in an unnecessary adaptations that incur potential resource costs and service disruption, which we term *penalty*.

Second, reactive adaptation lags behind current system conditions, and the degree of that lag depends on the sensitivity of the system sensors to present (versus historical) values of a system condition (e.g., CPU load, link bandwidth). If the system condition undergoes a dramatic and rapid shift, it may take numerous adaptation cycles for sensors to "catch up," resulting in more than one incremental adaptation change where a single adaptation might have sufficed. Again, this is problematic since each adaptation potentially incurs some penalty.

If a similar shift in system conditions recurs "seasonally"—once every *period* of time, such as every day at 8 AM—then the same undesirable pattern of incremental adaptations would repeat every period. (One workaround is to learn the seasonal pattern from historical data and predicate adaptations on time; however, this is a form of prediction.) In fact, executing adaptation while the system is under duress usually will take more time and is more likely to fail because of lack of resources. Having such prediction will help ensure sufficient resources are available for the adaptation.

Finally, knowledge of future availability of some required resource might result in a different adaptation choice that moves the system into a higher level of overall utility. To illustrate using a simplified Znn.com example, assume three levels of utility—*happy*, *somewhat happy*, *unhappy* – and three levels of values corresponding to resource conditions: *low*, *medium*, *high*. Assume that both high response time and zero service (i.e., no content) makes the customer unhappy, while low-fidelity content makes the customer somewhat happy. Assume further that an adaptation cycle takes one unit of time to effect its changes. We will represent the conditions of the system at a particular time-point with a tuple: (utility, response time, server load, available bandwidth, content fidelity). Now imagine a scenario lasting 3 time units, where Rainbow reacts to the conditions at time unit *1* by lowering the content fidelity:

0. (*happy* utility, *low* response time, *low* load, *high* available bandwidth, *high* fidelity)
1. (*unhappy*, *high*, *high*, *low*, *high*)
2. (*somewhat happy*, *medium*, *medium*, *low*, *low*)
3. (*somewhat happy*, *medium*, *medium*, *high*, *low*)

However, with perfect hindsight, knowing that the available bandwidth would recover to *high* might have led Rainbow to adapt by enlisting more servers to lower the average server load and to keep the fidelity high, thus achieving better overall utility:

3. (***happy***, *medium*, *medium*, *high*, *high*)

This example demonstrates how a reactive strategy of adaptation that optimizes instantaneous utility may often be sub-optimal over a long period of time. This deficiency results from two properties of reactive adaptation: (1) information used for decision making does not extend into the future, and (2) the planning horizon of the strategy is short and does not consider the effect of current decisions on future utility.

By analyzing its reactive nature, we have thus identified four opportunities for improving the current self-adaptation capabilities:

1. Preventing unnecessary self-adaptation

2. Reducing disruption from incremental adaptations.
3. Enabling pre-adaptation to seasonal behavior.
4. Improving overall choice of adaptation.

These opportunities for improving self-adaptation highlight the need for predictive information, particularly predictions of resources the target system environment. In the following section, we characterize a number of different kinds of prediction and types of information that are amenable to prediction.

3 Resource Prediction

In the previous section we identified four opportunities for using prediction to improve self-adaptation of systems. For the purpose of this chapter, *prediction* is an informed estimation of the future random values of a system or environment variable, e.g., the future available level of some resource required by the system. By leveraging predictive information, a self-adapting system is able to analyze adaptation alternatives slightly, or even significantly, ahead of real-time, make forward-looking decisions based on those predictions, and potentially improve the performance according to some objective metric. In this section, we describe the types of prediction that we use, discuss their applicability and limitations, and then describe a generic prediction framework that was developed for use in a ubiquitous computing context, but which can be co-opted for use within Rainbow.

Poladian defined and described an anticipatory model of self-adaptation in the context of a ubiquitous computing system that makes resource allocation decisions based on predictions of three inputs: (1) predictions of user's tasks, e.g., what type of applications the user needs and for how long, (2) predictions of resource demand by resource- and fidelity-aware applications, and (3) predictions of the available supply of resources such as network bandwidth and battery. He developed a calculus and framework that can synthesize different categories of prediction about a resource to produce a single combined predictive value. The types of predictive models that can be synthesized with this approach are: (1) *linear recent history*, which is a kind of predictor that uses recent history and a linear time-series model; we use autoregressive moving average (ARMA) models for this kind of resource prediction, which is consistent with [7]. (2) *Relative move,* which models seasonal variations in resource availability (e.g., knowing that network usage will be high at the beginning of a work day). (3) *bounding*, which specifies the maximum and minimum values of a resource for a union of time intervals (for example, knowing that bandwidth cannot be above 10Mbps). In this chapter, we are concerned with how to integrate the prediction architecture with a self-adaptive system, rather than the particular models of prediction used. For details of the types of predictive models, and the calculus for combining them, we refer readers to [20,21].

Because predictions are rarely perfect, a model of prediction must be prepared to address *uncertainty*. Broadly, *uncertainty* describes both measurement and estimation error when making predictions. Consequently, we differentiate between two types of uncertainty. The first type of uncertainty arises when estimating future, random values

of variables. One familiar example of such uncertainty is forecasting tomorrow's weather. Predicting (forecasting) tomorrow's temperature is generally imprecise, and a good prediction would provide an estimate for the uncertainty (error) in the forecast. Moreover, the error increases the further into the future one is predicting. Examples from computer systems include predicting the number of clients connected to the system or the available supply of network bandwidth in ten minutes. The second type of uncertainty arises when measuring the magnitude of past and present values of variables. An example from the physical sciences is the measurement of voltage. Here the uncertainty (error) is the result of imprecision, rather than randomness, that can only be resolved by waiting until some future time. An example from computer systems includes measuring the current available bandwidth between two network nodes.

Prediction and uncertainty in the context of self-adaptive systems must be modeled and addressed together. Typically, making predictions requires a statistical model that estimates (calculates) future values of a variable based on available information to the system. The uncertainty in the prediction is a rigorous description of the predictive error based upon that statistical model. In other words, prediction and uncertainty are described by the predictive distribution of the variable being estimated, conditional on all available data, e.g., the past values of the variable as well as the past values of the prediction errors and any other information.

The types of prediction models and the way of combining them can be extended to a certain class of self-adaptive systems that (a) monitor and predict resource availability, and (b) make resource allocation decisions as part of self-adaptive behavior. Typically, such systems are concerned with measuring or estimating both the *demand* for computational resources by the system under consideration and the *supply* of resources available to that system. In practice, the demand and the supply might be dependent. Therefore, it is important to identify when those are interdependent and express the dependence. Essentially this means whether each critical resource in the environment of the self-adaptive system is shared among many systems or entirely dedicated to the system under consideration. If the resource is not under our control, then we can simply use the aggregate predictions of that resource where the future value of that resource is based on the historical values of that resource. However, if the resource is being managed wholly by the self-adaptive system, then the prediction is more complicated; we need to predict how each element under our control uses that resource. In either case, the predictive framework can be applied equally effectively.

The kinds of predictions that can be handled by the prediction framework are for resources that have historical data that can be analyzed statistically and that match our statistical model of the resource in question. For example, if the historical data fits a Poisson distribution then it is obviously not applicable for an ARMA predictor that assumes Gaussian distribution. So, predictions that assume uncertainty is normally distributed may fail to detect the arrival of a so-called "Slashdot effect," when a rapid increase of web clients are connected to the server due to a sudden surge in the popularity of the web server. This is especially the case if the historical data does not contain evidence of a Slashdot event.

Our approach to anticipatory adaptation is based on optimizing the match between system needs and the environment capabilities. In practice, finding such a match corresponds to maximizing system utility. Poladian's thesis defines an analytical model that formalizes the notion of utility for user's tasks and expresses automatic configu-

ration as a mathematical problem of maximizing the expected utility of the user from the running state of the environment under the constraints of the computing environment.

The analytical model provides a carefully crafted structure for the problem, allowing efficient runtime configuration algorithms to search the problem space for good solutions. That structure is used to define a configuration strategy for prediction that takes as input (1) the amount of historical information about the resource being predicted, (2) the temporal horizon of the decisions, and (3) treatment of uncertainty in the available information explicitly quantifying the uncertainty of future events and coping with uncertainty by planning for future changes.

Using this analytical model, Poladian designed and implemented a software infrastructure for automatic configuration with three important contributions: (1) a central component that makes near optimal configuration decisions, (2) a prediction framework that provides resource prediction on demand, and (3) a programming interface between the centralized decision maker and the prediction framework. The central decision-making component leverages the structure of the analytical models to implement efficient and near-optimal configuration algorithms. In particular, the framework consists of the following four types of components, the architecture of which is defined in Fig. 5. For each resource, there will be one instantiation of this framework.

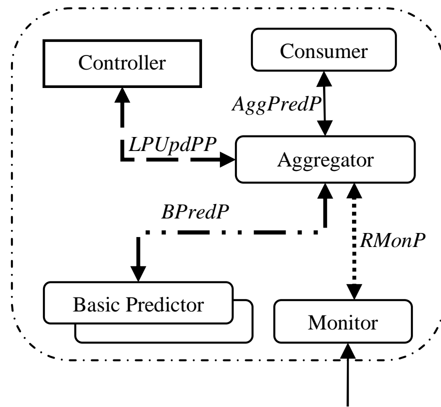

Fig. 3. The resource-prediction framework.

Aggregator: the centerpiece of the prediction framework, is responsible for combining information from all available Basic Predictors and calculating aggregate predictions. The Aggregator maintains an up-to-date list of currently available Basic Predictors. It aggregates the information from the predictors and produces a time series of predictions with increasing uncertainty further into the future.

Controller: allows setting the model parameters of the linear recent history predictor in the Aggregator. The model parameters are expected to be relatively stable over time, changing only infrequently. There is one Controller resource instance.

Basic Predictors: these components implement a wrapper around either known pattern or bounding predictors. Multiple Basic Predictors can be used. Upon startup, a Basic Predictor registers with the Aggregator. As new sources of predictions become available, additional Basic Predictors can be added to the framework,

Monitor: probes the environment for actual resource availability and provides periodic monitoring reports to the Aggregator. These monitored values correspond to the the historical values used by the predictors. A Monitor provides a uniform interface to the aggregator, encapsulating platform, network, and resource-specific details,

Consumer: the recipient and beneficiary of aggregate predictions. A Consumer is implemented by the coordinating entity of an adaptive resource management system. The prediction framework allows multiple concurrent Consumers to co-exist, each

with its own aggregate prediction session. The Consumer specifies prediction parameters to the Aggregator including the sampling window to make prediction observations and how far into the future to predict.

In summary, the resource prediction framework quantifies the future level of resource availability by combining predictive information from multiple sources. More details can be found in [21].

4 Incorporating Resource Predictions in Rainbow

Poladian's work on resource prediction is both practical in terms of algorithm speed, and useful in terms of manageable parameter space. In this section, we show how resource predictions can be incorporated into the Rainbow self-adaptation framework. Rainbow must satisfy the following requirements of Poladian's framework:

1. *Utility:* to evaluate the quality of the various possible adaptations on the system. Rainbow has a notion of utility as a central concept for strategy selection;
2. *Penalty*: to quantify costs of performing adaptations. If there is no penalty associated with adapting the system, this would obviate the need for using prediction – we will do a much better job with a reactive approach. In Rainbow, the penalties reflect the impact of temporary disruptions to system utility and the also time it takes to propagate changes throughout the system; and
3. *Historical information:* to facilitate prediction, past observed values need to be fed to the prediction framework.

4.1 Integration Points to Make Predictive Information Available

In Rainbow resource predictions can provide additional leverage in evaluating and choosing between alternate strategies of adaptation. For example, by knowing the probability that the available level of a critical resource, such as bandwidth, will be below a certain threshold 5 minutes from now, Rainbow can choose a strategy that quiesces lower priority client sessions so that the remaining client requests will continue to be satisfied within a tolerable latency. If, on the other hand, the probability is high that the bandwidth will be restored to levels that will naturally bring the system back within its desired state, Rainbow can choose to reduce the fidelity of some or all of the client sessions. Rainbow can even choose to do nothing.

To leverage resource predictions, it is important to consider how predictive information adds to the existing information flow of adaptation decisions in the Rainbow framework. The following points in Rainbow are potential sites for integrating resource predictions:

- Monitoring: predictor gauges
- Detection: prediction of architectural properties
- Strategy: conditions based on predicted value and actions with time cost
- Effector: addition or removal of prediction data streams

Monitoring: To make adaptation decisions, Rainbow reads gauge output to determine target-system conditions. We can integrate resource predictions in Rainbow by encapsulating, as gauges, instances of the entire prediction runtime from Poladian's system. It provides output to the gauge bus consistent with the gauge infrastructure API. Rainbow uses the standard gauge control interface to configure parameters of the prediction runtime. The gauge performs the role of the consumer, providing parameters for the prediction, then processing the time series returned from the aggregator to produce a single predicted value for one future time, as requested by Rainbow.

Because uncertainty is inherent in resource prediction, we must incorporate the probability of error in a predicted measurement, as supplied by predictors. We can choose to ignore predicted measurements and fallback to current measurements when the confidence level is below some threshold. We can also incorporate confidence level directly in utility computation to give lower consideration to strategies that use low-confidence predictive information.

Detection: Rainbow uses architectural constraints to identify opportunities for adaptation. Conditions based on predicted resource states, such as the anticipated load in the next 500 milliseconds, may indicate opportunities for adaptation. Thus, architectural constraints should support predicates over predicted values of architectural properties, perhaps in the form of a supplied architectural function, such as predictedProperty(p : Property, dur : int) : float (similarly for functions providing basic statistical operations, e.g., max/min/average). The predictedProperty() function returns the value of the architectural property identified by p, at a time point dur milliseconds from now. Recall that gauges are associated with specific architectural properties to update their values. So the function can compute predicted values by querying the predictor gauge mapped to the requested property.

Strategy: At adaptation time, Rainbow uses current system conditions (reflected in the model) to score and select strategies based on their expected utility. A strategy has two important ingredients: system *conditions* and adaptation *actions*. System conditions are used to (a) determine the applicability of strategies during strategy selection and (b) decide the next adaptation step during strategy execution. Adaptation actions change the target system to move the system toward a better state. New capabilities are required in the mechanisms for strategy selection, applicability condition, and actions to incorporate resource predictions.

An example strategy to reduce system response time is shown in Fig. 4, specified in Rainbow's adaptation language. The function defined on line 1, cPredViolation(), uses the architectural function predictedProperty() to compute client experienced response time at some future time, specified by dur. (cViolation defines the same predicate without using a predicted value.) Line 4 shows the use of this predicted value to deter-

```
01  define boolean cPredViolation (dur : int)=
      exists c : T.ClientT in M.components |
      Model.predictedProperty(c.experRespTime,
      dur) > M.MAX_RESPTIME;
02  ...
03  strategy VariedReduceResponseTime
04    [ cViolation && cPredViolation(self.dur) ] {
05      t0: (cViolation) -> enlistServers(1)
                                   @[1000 /*ms*/] {
06        t1: (!cViolation) -> done;
07        t2: (cViolation) -> lowerFidelity(2, 100)
                                   @[3000 /*ms*/] {
08          t2a: (!cViolation) -> done;
09          t2b: (default) -> TNULL;    // give up
10  } } }
```

Fig. 4. Sample snippet of an adaptation strategy.

mine the applicability of this strategy, in this case, when the client experienced response time is above threshold now and in the future. Lines 5-9 specify what the strategy does. In this case, it first enlists a server. Failing that, it then lowers the fidelity. And if that doesn't work, it gives up.

Timing plays a crucial role in prediction. We add the ability to calculate forward-looking expected utilities based on future system conditions. We augment Rainbow with the notion of *future variable value* so that a strategy can specify dependency on the future value of a condition. An adaptation action takes some time to execute, and estimating this duration is needed to determine how far into the future to predict. So, using settling time information specified in strategies (see @[ms] in Fig. 4, lines 5 and 7), Rainbow estimates the amount of *time* that a strategy would take to execute successfully. It then measures actual execution times to improve estimation.

For prediction to improve the performance of Rainbow, recall that there needs to be some cost, or *penalty,* to doing a particular adaptation. To capture this, we model penalty as a separate utility dimension, called disruption, that can be applied in utility-based strategy selection like other dimensions, such as average response time (see Table 1). There are two parts to disruption: one is how jarring it is to the user, and the other is how long the user is disrupted. We collect information about the disruption level in the same way as other dimensions, specified as part of the strategy specification. The second one we track automatically by measuring how long it takes to execute an adaptation step.

Effector: Finally, changes to the target system, particularly changes that add or remove resource components, will likely have significant effects on resource predictions. Therefore, we rely on system-level effectors to be augmented so that, when adding or removing system elements with associated resources, the effectors also take care of the addition or removal of the corresponding prediction data streams. Additionally, because prediction usually requires a series of input before the first output of predictive data, gauges may have to be coordinated with the addition of prediction data streams to produce useful output immediately.

4.2 Illustration of Rainbow with Resource Predictions

To illustrate resource predictions in Rainbow, let us revisit the Znn.com example to examine in more detail the four scenarios of prediction introduced in Section 2.3. Recall that in the Znn.com example, the customers care about quick response time and high content fidelity for their news requests. While aware of customer preferences on content fidelity, Znn.com as the provider is constrained by infrastructure provisioning costs. We also consider service disruption as a penalty of performing an adaptation: avoiding penalties is important to improving overall system utility, which is a major benefit to having predictive information.

Accordingly, we define four quality dimensions and determine the corresponding measurable properties in the target system. We capture each dimension as a discrete set of values (for example, we use an ordinal scale of 1 to 5 to express the degree of disruption). We then elicit from the service providers the utility values and preferences for these dimensions, summarized in Table 1.

Table 1. Znn.com quality dimensions and utility preferences

Label	Description	Architectural Property	Utility Function	Weight
uR	Avg Response Time	ClientT.experRespTime	((low,1), (med,0.5), (high,0))	25%
uF	Avg Content Fidelity	ServerT.fidelity	((textual,0), (multi-media,1))	10%
uC	Avg Budget	ServerT.cost	((within,1), (over,0))	15%
uD	Disruption	ServerT. droppedReqs	((1,0.8), (2,0.6), (3,0.4), (4,0.2), (5,0))	50%

A rule specifies the acceptable bound of request-response latencies experienced by a client: exceeding the threshold indicates a problem. A set of operators correspond to available effectors in Znn.com to enlist or remove servers, or to change content fidelity. We define a number of adaptation strategies for Znn.com and specify cost-benefit attribute vectors, not shown here, that specify the impact of each strategy to the four quality dimensions. For example, strategy VariedReduceResponseTime is expected to lower response time and fidelity level, not affect cost, and incur some disruption.

We now consider how prediction could improve Rainbow's choices of adaptation for the four opportunities outlined in Sec. 2.3. For evaluation, we set up Znn.com in a simulation environment that allows us to experiment with prediction-enabling design points in Rainbow's Architecture Layer (cf. Fig. 2). The states of Znn.com are simulated using an M/M/k queuing model. The simulation environment acts as gauges that update corresponding Znn.com architectural properties in Rainbow. This setup enables prediction of future states to an arbitrary precision.

Scenario 1: avoiding unnecessary adaptation

In the first scenario, if a client experiences an above-threshold request-response time for only 500 ms, but the chosen adaptation requires at least one second to complete, this adaptation is unnecessary. Avoiding adaptation requires knowing the predicted request-response time (using the architectural function predictedProperty()) and the estimated execution time of an adaptation strategy, which Rainbow collects.

To evaluate how well prediction improves overall system utility in this scenario, we designed two Znn.com configurations, one in which the bandwidth drops briefly, and another in which incoming requests (load) spike briefly. The data is summarized in Table 2. In both cases, Rainbow with prediction successfully avoided making unnecessary adaptations, improving the normalized accrued utility over no prediction by 2.5% in the transient bandwidth-drop case, and 15.7% in the transient peak-load case. The much greater improvement in the second case can be attributed to the high level of disruption incurred by the strategy that is unnecessarily invoked without future knowledge This outcome underscores the role of *penalty* in determining whether prediction is useful. We discuss some choices of prediction usage in Section 4.3.

Scenario 2: reducing incremental disruptions

In the second scenario, Znn.com experiences a dramatic increase in client requests, ramped up over seconds to minutes. In reaction, Rainbow provisions by invoking a strategy that adds one server. However, by the time the server is added, the request load has surpassed the capacity of the added server, so Rainbow adds another server in response. This gradual adaptation is undesirable because it disrupts the system

multiple times. Eliminating this ramp-up requires knowing the peak of the ramp-up and computing cost-benefit attributes based on input arguments to an adaptation step (e.g., *k* in enlistServers(*k*)).

To evaluate this scenario, we designed a Znn.com configuration that ramps up requests over four seconds. We added a *leap* strategy similar to VariedReduceResponseTime but enlists 3 servers in one step. We then configured Rainbow to compute utilities that look five seconds ahead, and compute the load at its peak. Rainbow successfully selected the leap strategy and showed a 4.9% improvement in AU.

The results of these experiments show that there is improvement when using predictive information. Perhaps not surprisingly, the most improvement is achieved when the potential disruption to the user is high. For the other cases, the room for improvement is not as great, but our numbers are significant when measured against the available margin for attaining perfect utility.

Table 2. Summary of data from 3 experiments (each averaged over 30 trials)

Scenario Configuration	**Normalized Accrued Utility (AU)**		**ΔAU**	**Improved**
	No Prediction	With Prediction		
1: transient bandwidth-drop	0.889	0.911	0.022	2.5%
1: transient peak-load	0.731	0.846	0.115	15.7%
2: ramp-up to peak load	0.734	0.770	0.036	4.9%

Additional scenarios: seasonal pre-adaptation and choosing better adaptations
We have shown two scenarios that exercised the new capabilities added to Rainbow to incorporate predictive information, with supporting data from experiments. We now consider two other scenarios that use the same set of capabilities; for these we have not performed additional experiments.

In a third scenario, Znn.com periodically experiences a significant increase in client requests at 9 AM every Monday through Friday. Reacting to the increase each time it occurs is undesirable because the adaptation potentially disrupts the system and adds stress to a system already under load. In contrast, pre-adapting has the benefit of reducing disruption while introducing system *slack* to prepare for the upcoming load. Pre-adapting for seasonal behavior requires detecting seasonal patterns, which can be provided by predictors in Vahe's framework. Then, by adding an architectural constraint that checks for predicted load at fixed future time points, configuring utility computation to look ahead to the same time, and specifying a strategy that is applicable for violation at that future time point, Rainbow can seasonally pre-adapt.

A fourth scenario is already described in Section 2.3, where a client experiences an above-threshold request-response time due to increased visitor traffic, coupled with a transient drop in available bandwidth. Given the low bandwidth and a choice between the a strategy to lower fideltiy and another to enlist more servers, Rainbow chooses the former to use less bandwidth while fulfilling the increased request load. However, when the available bandwidth recovers shortly afterward, Rainbow would then adapt again to restore the content fidelity and perhaps also enlarge the server pool if traffic remains high. Thus, Rainbow's reaction results in at least one additional disruption and an overall lower system utility. With advanced knowledge that the bandwidth drop is transient (as in scenario 1), Rainbow would have chosen to enlist servers.

4.3 Deciding When to use Predictive Information

Once predictive information is available for use in self-adaptation, the questions still remain of when and how to use the information in the decision process. In our application of predictive information, we encountered the following design choices, which we have addressed in a variety of ways.

How far into the future do we look ahead? The predictive framework requires parameterization for how far ahead to predict a resource property. The predictive framework actually returns a time series of values, but to make use of this information in Rainbow, we must pick one particular value. The choice of this depends on the context. For example, in Scenario 1 where we are trying to decide whether to avoid an adaption, a reasonable choice is to use a duration equivalent to the estimated time of completing the adaptation. For Scenario 2 on the other hand, the look ahead could be far longer than the duration of adaptation. Note that by using estimated completion time to choose how far into the future to look, we are comparing different prediction ranges for different strategies in a single adaptation cycle. An alternative is to look ahead to the same time in the future, perhaps by using the maximum completion time of all strategies under consideration.

Should predictive information be used at strategy selection time, utility evaluation time, strategy execution time, or a combination of these? There are several steps in Rainbow's process of selecting a repair strategy: 1) decide the set of strategies that may fix a problem; 2) determine which strategy is the best to use; and 3) execute the chosen strategy. Predicted information can be used in Rainbow at any of these times. For example, the strategy in Fig. 4 uses predicted information in step 1. In line 4, we are checking if response time is high now and in the future. If the condition is transient the strategy will not be chosen, and so there is no need to use prediction in lines 5-9 (strategy execution time). Alternatively, to anticipate seasonal changes, the strategy writer would write the strategy to consider only predictions in line 4, and also to use prediction in lines 5-9. Rainbow gives the strategy writer the power to decide how and when to use the prediction. Currently, in Rainbow, the second step (using prediction in the utility calculation) is provided as a parameter to the framework, because there is no way in the strategy language to refer to this. We are investigating a more agile way to specify the use of prediction in this case.

How much weight should be given to the penalty dimension? When we experimented with a 10% weight for the penalty dimension, the first configuration yielded a utility improvement of 0.4%, whereas a 50% weight yielded 2.4% improvement. This data reinforced Poladian's results that anticipatory adaptation yields increasing gains at penalty levels above 7%. On the flip side, a penalty weight above 50% makes it difficult to distinguish the relative importance of the other utility dimensions. A sweet spot should be found between 10 and 50%.

5 Related Work

To date, several dynamic software architecture-based adaptation approaches and frameworks have been proposed and developed [12, 19]. Related approaches focus on

formalism and modeling, mechanisms for adaptation, or distribution and decentralization of control. These include Darwin with π-calculus semantics to specify distributed systems [16], ArchWare with architectural reflection and dynamic co-evolution [17], Weaves for construction and analysis of data-flow systems [13], ArchStudio for self-adaptation of C2 hierarchical publish-subscribe systems [6], Plastik targeting performance properties [1], and CASA for resource availability concerns in mobile network environments [18]. These approaches share a few common characteristics: They generally apply a closed-loop control, use an architecture model for reasoning about the target system, assume certain structures in the target system, and adapt for a fixed set of quality attributes.

Notable in industry, IBM's Autonomic Computing tackles the challenges of emergent autonomic behavior with the MAPE control loop—to monitor, analyze, plan, and execute changes for self-management. The AC toolkit provides consoles and tools to diagnose problems and engineer autonomic systems. We apply a similar approach.

One of the differentiators of this work from prior self-adaptive systems is the use of resource prediction. The anticipatory strategy uses predictions of the future values of input variables to make forward-looking decisions about adaptation selection. Forward-looking approaches have been proposed and used in other domains. For example, the online stochastic combinatorial optimization approach is similar to our anticipatory strategy [2,14]. Various combinatorial optimization problems such as optimal vehicle dispatch and network packet routing are solved by leveraging probabilistic priors of the future values of problem inputs. There is equivalence between the algorithms for automatic configuration in this chapter and the algorithms described in [14]. The Active Virtual Network Management Prediction System uses simulation models running ahead of real time to predict resource demand among network nodes. Such predictions can be used to allocate network capacity in anticipation of demand increase, and to ensure adequate quality of service to different network flows [9]. Our work shares theoretical foundations with these, but the problem domains are different.

There is a body of work that uses various kinds of prediction to improve self-adaptation. For example, Clockwork [22] introduces the concept of *predictive autonomicity* that uses statistical modeling to forecast cyclic variations in system load and uses these predictions to reconfigure systems in anticipation of need. They prescribe a method for implementing a predictive autonomic system. In spirit, we share the same steps for incorporating predictive information. However, our notion of controllable parameters are enriched with strategies and utility preferences, and we use predicted information in strategy selection. Solomon [23] uses predictions about workload to adapt the control component of an autonomic system to be more suited to that workload. For example, if the workload is linear, then simple thresholding can be used in the controller, but if the workload on the system changes to be Gaussian, then a more sophisticated statistical controller based on Kalman filters is swapped in to manage the system. Their adaptation layer shares the same principle components as Rainbow, with the selection of controllers analogous to selection of strategies. They show encouraging results in using predicted information for Gaussian workloads to provision servers. This is one of many types of prediction sources that could be incorporated into our prediction framework.

Rather than using auto-regressive techniques to predict resource availability, Lu [15] uses knowledge about the domain being controlled to predict behavior. They use

queuing-theoretic models in the domain of web servers to infer expected delays directly from input load. Again, this is another form of predictive model that could theoretically be incorporated as a Basic Predictor in our framework, although it is of a type that we have not fully considered.

6 Conclusion and Future Work

In this chapter, we presented an approach to enhance architecture-based self-adaptation through anticipatory prediction of future resource availability. The approach uses a framework that combines various forms of prediction (statistical, bounded, and seasonal) in a practical manner that can be applied to a variety of circumstances. We have argued that self-adaptive systems can take advantage of prediction to improve the choice of adaptations and to reduce disruption to the system. We gave specific consideration to the changes needed to incorporate predictions into one reactive architecture-based self-adaption system, Rainbow. We conducted several experiments that show improvement in the adaptation when predicition is used, and discussed how we addressed some issues that we encountered doing the integration.

In future work, we would like to better quantify the types of resources that can be predicted and would be useful in realistic circumstances. For example, other types of resources to be considered beyond bandwidth are power consumption, memory usage and CPU load. We would like to give more guidance to adaptation writers about when and how to use prediction. We also would like to verify the results discussed in this chapter through additional experimentation and application to real systems.

Acknowledgments. This research was funded in part by the National Science Foundation Grants ITR-0086003, CCR-0205266, CCF-0438929, CNS-0613823, by the Sloan Software Industry Center at Carnegie Mellon, by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and by DARPA grant N66001-99-2-8918. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the US government or any other entity.

References

1. 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.
2. Bent, R. and van Hentenryck, P.: Regrets only! Online stochastic optimization under time constraints. In: Proc. 19th AAAI (2004).
3. Cheng, S.-W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation, Ph.D. Thesis, TR CMU-ISR-08-113, Carnegie Mellon University School of Computer Science, May (2008).
4. Cheng, S.-W., Garlan, D., Schmerl B: Making Self-Adaptation and Engineering Reality. In Babaoghu, O et al. (eds), *Proc. Conference on Self-Star Properties in Complex Information Systems,* LNCS (3460), 2005.

5. Clements, P., et al.: Documenting Software Architecture: Views and Beyond, Pearson Education (2003).
6. Dashofy, E.M., van der Hoek, A., and Taylor, R.N.: Towards architecture-based self-healing systems. In: Garlan et al. [10], 21–26 (2002)
7. Dinda, P., O'Halloran, D. Host Load Prediction Using Linear Models. *Cluster Computing*, 3:4, 2000.
8. Frye, C.: Self-healing systems. In: Appl. Dev. Trends, September, 29--34 (2003)
9. Galtier, V., et al.: Predicting resource demand in heterogeneous active networks. In: Proc. MILCOM (2001)
10. Garlan, D., Kramer, J., and Wolf, A. (eds.): Proc. 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), New York, NY, USA, November 18--19, ACM Press (2002)
11. Georgiadis, I., Magee, J., and Kramer, J.: Self-organizing software architectures for distributed systems. In: Garlan et al. [10], 33–38 (2002)
12. Ghosh, D., Sharman, R., Rao, H.R., and Upadhyaya, S.: Self-healing systems - survey and synthesis. In: Decision Support System, 42(4), 2164--2185 (2007)
13. Gorlick, M.M. and Razouk, R.R.: Using Weaves for software construction and analysis. In: Proc. 13th International Conf. of Software Engineering, 23--34, Los Alamitos, CA, USA, May, IEEE Computer Society Press (1991)
14. Hentenryck, P., et al. Online stochastic optimization under time constraints. In http://www.cs.brown.edu/people/pvh/aor5.pdf, working paper, last accessed April (2008)
15. Lu, Y., Abdelzaher, T., Lu, C., Sha, L., and Liu, X. Feedback Control with Queuing-Theoretic Prediction for Relative Delay Guarantees in Web Servers. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.
16. 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.
17. 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.
18. 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.
19. Oreizy, P., et al.: An architecture-based approach to self-adaptive software. In: IEEE Intelligent Systems, 14(3), 54--62, May--June (1999)
20. Poladian, V., Garlan, D., Shaw, M., Schmerl, B., Sousa, J.P., and Satyanarayanan, M. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In *Proc. 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO-2007),* pp. 214-223, July 2007.
21. Poladian, V.: Tailoring Configuration to User's Tasks under Uncertainty, Ph.D. Thesis, TR CMU-CS-08-121, Carnegie Mellon University School of Computer Science, May (2008)
22. Russel, L., Morgan, S. And Chron, E. Clockwork: A new movement in autonomic systems. *IBM Systems Journal*, 42:1, 2003.
23. Solomon, B., Ionescu, D., Litoiu, M., Mihaescu, M. A Real-Time Adaptive Control of Autonomic Computing Environments. In *Proc. 4th International Information and Telecommunication Technologies Symposium (U2TS'2006),* pp. 94-103, Dec. 2006.
24. Sousa, J.P.: Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments, Ph.D. Thesis, TR CMU-CS-05-123, Carnegie Mellon University School of Computer Science, (2005)
25. Sztajnberg A., and Loques, O.: Describing and deploying self-adaptive applications. In: Proc. 1st Latin American Autonomic Computing Symposium, July 14--20 (2006)