Software Architecture-Based Adaptation for Pervasive Systems

Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu

School of Computer Science, Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213 USA
Phone: 1 412 268 5056
{zensoul,garlan,schmerl,jpsousa,sprite,prs,hnn}@cs.cmu.edu

Abstract. An important requirement for pervasive computing systems is the ability to adapt at runtime to handle varying resources, user mobility, changing user needs, and system faults. In this paper we describe an approach in which dynamic adaptation is supported by the use of software architectural models to monitor an application and guide dynamic changes to it. The use of externalized models permits one to make reconfiguration decisions based on a global perspective of the running system, apply analytic models to determine correct repair strategies, and gauge the effectiveness of repair through continuous system monitoring. We illustrate the application of this idea to pervasive computing systems, focusing on the need to adapt based on performance-related criteria and models.

1 Introduction

An important requirement for pervasive computing systems is the ability to adapt themselves at runtime to handle such things as user mobility, resource variability, changing user needs, and system faults. In the past, systems that supported self-adaptation were rare, confined mostly to domains like telecommunications switches or deep space control software, where taking a system down for upgrades was not an option, and where human intervention was not always feasible. However, in a pervasive computing world more and more systems have this requirement, because they must continue to run with only minimal human oversight, and cope with variable resources as a user moves from one environment to another (bandwidth, server availability, etc.), system faults (servers and networks going down, failure of external components, etc.), and changing user priorities (high-fidelity video streams at one moment, low fidelity at another, etc.).

Traditionally system self-repair has been handled within the application, and at the code level. For example, applications typically use generic mechanisms such as exception handling or timeouts to trigger application-specific responses to an observed fault or system anomaly. Such mechanisms have the attraction that they can trap an error at the moment of detection, and are well-supported by modern programming languages (e.g., Java exceptions) and runtime libraries (e.g., timeouts for RPC). However, they suffer from the problem that it can be difficult to determine the true source of the problem, and hence the kind of remedial action required. Moreover, while they can trap errors, they are not well-suited to recognizing "softer" system anomalies, such as gradual degradation of performance over some communication path.

Recently several researchers have proposed an alternative approach in which system models – and in particular, software architectural models – are maintained at runtime and used as a basis for system reconfiguration and repair [25]. An architectural model of a system is one in which the overall structure of a running system is captured as a composition of coarse-grained interacting components [28]. As a basis for self-repair the use of architectural models has a number of nice properties: An architectural model can provide a global perspective on the system allowing one to determine non-local changes to achieve some property. Architectural models can make "integrity" constraints explicit, helping to ensure the validity of any change. By "externalizing" the monitoring and adaptation of a system using architectural models, it is possible to engineer adaptation mechanisms, infrastructure and policies independent of any particular application, thereby reducing the cost and improving the effectiveness of adding self-adaptation to new systems.

In this paper we illustrate how architecture-based adaptation can be applied to pervasive computing systems. Specifically, we show how to use the approach to support adaptation of applications in a pervasive computing environment. This pervasive environment consists of a set of mobile users accessing shared information through a variety of devices. These devices communicate over a heterogeneous communications infrastructure.

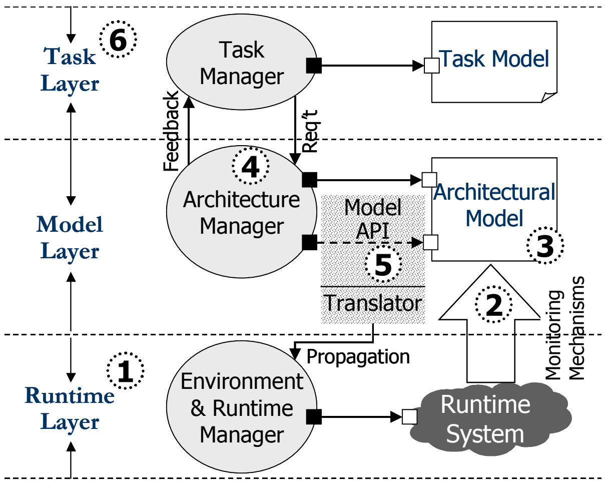


Fig. 1. Adaptation Framework

2 Overview of Approach

Our approach is based on the 3-layer view illustrated in Figure 1. The *Runtime Layer* is responsible for observing a system's runtime properties and performing low-level operations to adapt the system. It consists of the system itself, together with its operating environment (networks, processors, I/O devices, communications links, etc.) (1). Observed runtime information is propagated upwards using a monitoring infrastructure that condenses, filters, and abstracts those observations in order to render that information in architecture-relevant terms (2).

The *Model Layer* is responsible for interpreting observed system behavior in terms of higher-level, and more easily analyzed, properties. It forms the centerpiece of the approach, consisting of one or more architectural models of the system (3), together with respective architecture managers (4) that determine whether a system's runtime behavior is within the envelope of acceptable ranges. An architecture manager includes a constraint checker and a repair handler. The former determines when architectural constraints are violated; the latter determines how to adapt the system. Repairs are propagated down to the running system (5).

The *Task Layer* is responsible for determining the quality of service requirements for the task(s). A task is a high-level representation of a user's computational needs, and indicates the services required, as well as the desired performance profile for those services. These profiles in turn determine the range of behavior permissible at an architectural level.

To illustrate how the approach works, consider a set of mobile users interacting with a pervasive environment, each user currently performing one or more tasks that require access to shared information. We will assume that this shared information is provided by a set of server groups distributed over a pervasive network, as illustrated in Figure 2(a). Each server group consists of a set of replicated servers (Figure 2(b)), and maintains a queue of requests, which are handled in FIFO order by the servers in the server group. Individual servers send their results back directly to the requesting user.

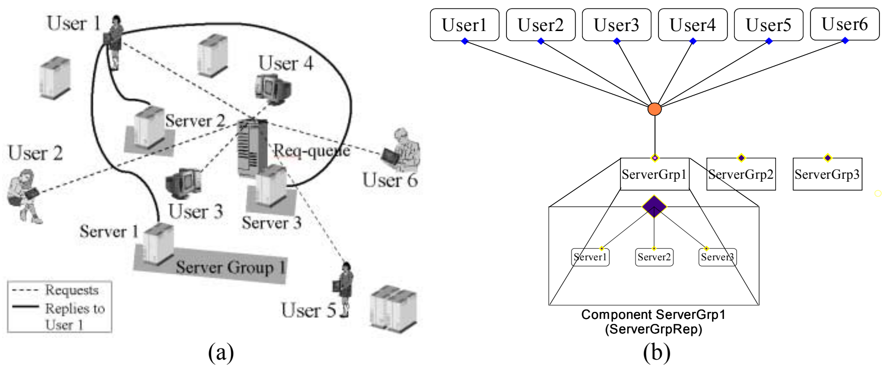


Fig. 2. Deployment Architecture (a) and Software Architecture (b) of the Example System

The pervasive computing environment that manages this overall infrastructure needs to make sure that two inter-related system qualities are maintained. First, to guarantee the quality of service for each user, the request-response latency for users must be under a certain threshold, which may vary depending on the task and user. Second, to keep costs down, the set of currently active servers should be kept to a minimum, subject to the first constraint.

Achieving these goals requires cooperation from three levels. The *Task Layer* has knowledge of the kind of information a user requires and the quality of service requirements for retrieving this information. This knowledge feeds into the *Model Layer*, so that relevant analyses can be performed to determine the appropriate configuration when a new task is created. The *Model Layer* then makes changes through the *Runtime Layer*, to the executing system to fulfill those requirements.

Establishing the correct configuration for the system only when a task is created, however, is not sufficient in a pervasive computing environment, since resources and requirements change dynamically. For example, suppose that some user's task requires her to review a set of images and select some of them to be included in a report. Suppose that initially this user is carrying out the task on a PDA, which communicates over a wireless network, and which can only display low-resolution grayscale images. As the user moves through the environment, her PDA may move from a wireless cell that has an access point getting good bandwidth to a server group to a cell that is not. In this case, the environment may need to locate another server group with a better bandwidth and move her requests to that server group. This change of resources should be sensed automatically and the reconfiguration done transparently, so that the user is not unnecessarily distracted. Furthermore, this same user might later move into a resource-rich environment that contains a high-resolution color display. The task layer may then want to change the user's bandwidth requirements so that she can view larger images on this screen. These new bandwidth requirements may force a change in the *Model Layer*, which will invoke a concomitant change in the implementation.

The approach outlined above has a number of distinct advantages for the systems builder over current approaches that hardwire adaptation mechanisms into the components of the application. First, the use of architectural models permits non-local properties to be observed, and non-local adaptations to be effected. For example, suitable monitoring mechanisms can keep track of aggregate average behavior of a set of components. Second, formal architectural models permit the application of analytical methods for deriving sound repair strategies. For example, a queuing-theoretic analysis of performance can indicate possible points of adaptation for a performance-driven application. Third, externalized adaptation (via architectural models) has several important engineering benefits: adaptation mechanisms can be more easily extended; they can be studied and reasoned about independently of the monitored applications; they can exploit shared monitoring and adaptation infrastructure.

3 Architecture-Based Adaptation

The centerpiece of our approach is the use of stylized architectural models [26,28]. Although there are many proposed modeling languages and representation schemes for architectures, we adopt a simple scheme in which an architectural model is represented as a graph of interacting components. This is the core architectural representation scheme adopted by a number of architecture description languages, including

Acme [11], xADL [8], and SADL [23]. Nodes in the graph are termed components. They represent the principal computational elements and data stores of the system: clients, servers, databases, user interfaces, etc. Arcs are termed connectors, and represent the pathways of interaction between the components. A given connector may in general be realized in a running system by a complex base of middleware and distributed systems support. For example, in the software architecture illustrated in Figure 2(b), the server group, servers, and users are components. The connector includes the request queue and the network connections between users and servers.

To account for various behavioral properties of a system we allow elements in the graph to be annotated with extensible property lists. Properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). Properties associated with a component might define its core functionality, performance attributes (e.g., average time to process a request, load, etc.) or reliability. In addition we associate with each architecture a set of constraints defined in a first-order predicate logic augmented with a set of primitives appropriate for architectural specification [22]. These constraints can be attached to components or connectors to express things like the fact that some property value must always lie between a given range of values.

In our system each architecture is identified with a particular architectural style. An architectural style defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed. Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [10,31,32]. Moreover, the notion of style often maps well to widely-used component integration infrastructures (such as EJB, HLA, CORBA), which prescribe the kinds of components allowed and the kinds of interactions that may take place between them.

One of the significant advantages of architectural descriptions is that they provide opportunities for analysis, including system consistency checking [3], conformance to architectural style constraints [1], conformance to quality attributes [7], and dependence analysis [30].

We can model our example using a client-server architectural style. The architectural style provides definitions for client, server, and server group components and the connections between them. Properties include those required for queuing-theoretic performance analysis, and integrity constraints include the necessity for each client to be connected to one and only one server group.

As mentioned in our example, the *Task Layer* sets the performance profile for the architecture. These profiles can be expressed as threshold constraints in the architecture. These constraints can then be checked dynamically to see if the system is functioning within bounds. In the context of our example, we desire each user to receive no more than some maximum latency. This can be expressed in the architecture as a constraint on each of the client's connections to the server group. In the architecture of our example, the constraint is of the form:

```
averageLatency < maxLatency.
```

This constraint appears on each client's connection, and needs to be evaluated dynamically. In our approach, the *Task Layer* sets the value for *maxLatency*; the *averageLatency* value is an observed value determined by monitoring.

3.1 Using Architectural Analysis to Guide System (Re)Configuration

As we argued above, one of the main benefits of using software architecture is that the level of abstraction gives us the ability to use analytical methods to evaluate properties of a system's architectural design. To illustrate how this works, consider our example, where we have modeled the application in a style amenable to M/M/m performance analysis [29]. The M/M indicates that the probability of a request arriving at component *s*, and the probability of component *s* finishing a request it is currently servicing, are assumed to be exponential distributions (also called "memoryless," independent of past events); requests are further assumed to be, at any point in time, either waiting in one component's queue, receiving service from one component, or traveling on one connector. The *m* indicates the replication of component *s*; that is, component *s* is not limited to representing a single server, but rather can represent a server group of *m* servers that are fed from a single queue. Given estimates for clients' request generation rates and servers' service times (the time that it takes to service one request), we can derive performance estimates for components.

Applying this M/M/m theory to the style used in our example tells us that with respect to the average latency for servicing user requests, the key design parameters in our style are (a) the replication factor *m* of servers within a server group, (b) the communication delay between clients and servers, (c) the arrival rate of client requests, and (d) the service time of servers within a server group. We can use performance analysis to decide (1) the number of replicated servers that must exist in a server group so that it is properly utilized, and (2) where server groups should be placed so that the bandwidth is sufficient to achieve the desired latency.

Given a particular service time and arrival rate, performance analysis of this model gives a range of possible values for server utilization, replication, latencies, and system response time. Say that the task layer for each user informs us that the arrival rate is 180 requests/sec, the average request size is 0.5KB, and the average response size is 20KB. Assume also that the server service time is between 10ms and 20ms. Given these values, then the performance analysis gives us the following bounds:

Initial server replication count= 3-5
Average Bandwidth = 10.5KB/sec

This analysis gives us parameters for a configuration of the architecture of the software that satisfies the above requirements. We use this information to configure the system to locate appropriate server groups, monitor the application to make sure it is in conformance with these requirements, and attempt to adapt the system transparently as the user moves about the environment.

If the *Task Layer* changes the requirements, for example when the user begins using a large display, the analysis is performed again to determine a satisfactory reconfiguration of the system. Again, this can be done transparently.

3.2 Using Architecture to Assist Adaptation

The representation schemes for architectures and analyses outlined above were originally created to support design-time development tools. As suggested above, these schemes and analyses need to be made available at runtime. This section discusses an augmentation to architectures that allows them to function as runtime adaptation mechanisms. This includes *adaptation operations*, based on the style of the architecture, to change an architectural model, and *repair strategies* that apply these operations to adapt the architecture. These operations need to be translated into operations on the runtime system. We consider the supporting runtime infrastructure needed to make this work in practice in Section 3.3.

3.2.1 Architecture Adaptation Operators

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

Given a particular architectural style, there will typically be a set of natural operators for changing an architectural configuration and querying for additional information. In the most generic case, architectures can provide primitive operators for adding and removing components and connections [24]. However, specific styles can often provide higher-level operators that exploit the restrictions in that style and the intended implementation base.

In terms of our example, we define the following operators:

addServer(): This operation is applied to a server group component and adds a new replicated server component to its representation, ensuring that the architecture is structurally valid.

move(*to:ServerGroupT*): This operation is applied to a client and deletes the role currently connecting the client to the connector that connects it to a server group and performs the necessary attachment to a connector that will connect it to the server group passed in as a parameter.

remove(): This operation is applied to a server and deletes the server from its containing server group. Furthermore, it changes the replication count on the server group and deletes the binding.

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

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

These operators reflect the style in question and the implementation base. First, from the nature of a server group, we get the operations for activating or deactivating a server within a group. Also, from the nature of the asynchronous request connectors, we get the operations for adapting the communication path between particular clients and server groups. Second, based on the knowledge of supported system change operations, outlined in Section 3.3.2, we have some confidence that the architectural operations are actually achievable in the executing system.

3.2.2 Architecture Repair Strategies

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

A repair strategy has two main functions: first to determine the cause of the problem, second to determine how to fix it. Thus the general form of a repair strategy is a sequence of repair tactics. Each repair tactic is guarded by a precondition that determines whether that tactic is applicable. The evaluation of a tactic's precondition will usually involve the examination of various properties of the architecture in order to pinpoint the problem and determine applicability. If it is applicable, the tactic executes a repair script that is written as an imperative program using the style-specific operators described above.

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

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

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

Illustrating this idea for our example, we can show how the repair strategy developed from the theoretical analysis. The equations for calculating latency for a service request, derived from [4], indicate that there are four contributing factors: 1) the connector delay, 2) the server replication count, 3) the average client request rate, and 4) the average server service time. Of these we have control over the first two. When the latency is high, we can decrease the connector delay or increase the server replication count to decrease the latency. Determining which tactic depends on whether the connector has a low bandwidth (inversely proportional to connector delay) or if the server group is heavily loaded (inversely proportional to replication). These two system properties form the preconditions to the tactics; we have thus developed a repair strategy with two tactics.

Figure 3 illustrates the repair strategy and tactics associated with a latency threshold constraint. Line 1 defines the constraint that the average latency must not be below the maximum latency set by the task requirements. Line 2 calls the repair strategy to be invoked if the constraint fails. The repair strategy in lines 4-14, *fixLatency*, consists of two tactics. The first tactic, defined in lines 16-26, handles the situation in which a server group is overloaded, identified by the precondition in lines 22-23. Its main action in lines 24-25 is to create a new server in any of the overloaded server groups. The second tactic, defined in lines 28-42, handles the situation in which high latency is due to communication delay, identified by the precondition in lines 30-31. It queries the architecture to find a server group that will yield a higher bandwidth connection in lines 35-36. In lines 37-39, if such a group exists it moves the client-server connector to use the new group. The repair strategy uses a policy in which it executes these two tactics sequentially: if the first tactic succeeds it commits the repair strategy; otherwise it executes the second. The strategy will abort if neither tactic succeeds, or if the second tactic finds that it cannot proceed since there are no suitable server groups to move the connection to.

```
invariant r : averageLatency <= maxLatency
!→ fixLatency(r);

strategy fixLatency (badRole : ClientRoleT)={
  let badClient : ClientT =
    select one cli : ClientT in self.Components |
      exists p : RequestT in cli.Ports |
        attached(badRole, r);
  if (fixServerLoad(badClient)) {
    commit repair; }
  else if (fixBandwidth(badClient,badRole) {
    commit repair; }
  else {abort ModelError;}
}

tactic fixServerLoad (client :ClientT) :boolean={
  let loadedServerGroups :set{ServerGroupT}=
    select sgrp:ServerGroupT in
        self.Components |
          connected(sgrp,client) and
          sgrp.load > maxServerLoad;
  if (size(loadedServerGroups) == 0)
    return false;
  foreach sGrp in loadedServerGroups {
    sgrp.addServer(); }
  return (size(loadedServerGroups)>0);

tactic  fixBandwidth(client:ClientT
                          role:ClientRoleT):boolean={
  if (role.bandwidth>=minBandwidth) {
    return false;}
  let oldSGrp: ServerGroupT =
    select one sGrp:ServerGroupT in
      self.Components |  connected (client,sGrp);
  let goodSGrp : ServerGroupT =
    findGoodSGrp(client,minBandwidth);
  if (goodSGrp != nil) {
    client.move (oldSGrp,goodSGrp);
    return true;
  } else {
    abort NoServerGroupFound;
}}
```

Fig. 3. Repair Strategy for High Latency

3.3 Bridging the Gap to Implementation

While the use of architectural models allows us to provide automated support for adaptation at an architectural level, through use of constraints, operators, and analytical methods, we must furthermore relate model changes to the real world. There are two aspects to this. The first is getting information out of the executing system so we can determine when architectural constraints are violated. The second is propagating architectural repairs into the system itself.

3.3.1 Monitoring

In order to provide a bridge from system level behavior to architecturally-relevant observations, we have defined a three-level approach illustrated in Figure 4. This monitoring infrastructure is described in more detail elsewhere [12]: here we summarize the main features.

The lowest level is a set of *probes*, which are "deployed" in the target system or physical environment. Probes monitor the system and announce observations via a "probe bus." We can use off-the-shelf monitoring components (such as Remos [19]) and write wrappers to turn them into probes, or write custom probes. At the second level a set of *gauges* consume and interpret lower-level probe measurements in terms of higher-level model properties. Like probes, gauges disseminate information via a "gauge reporting bus." The top-level entities in Figure 4 are *gauge consumers*, which consume information disseminated by gauges. Such information can be used, for example, to update an abstraction/model, to make system repair decisions, to display warnings and alerts to system users, or to show the current status of the running system.

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

For instance, in the example above we are concerned with the average latency of client requests. To monitor this property, we must associate a gauge with the *averageLatency* property of each client role. Each latency gauge in turn deploys a probe into the implementation that monitors the timing of reply-request pairs. When it receives such monitored values it averages them over some window, updating the latency property in the architecture model when it changes. In addition to this gauge, we are also guided by the repair tactics to place gauges that measure the bandwidth between the client and the server group and also to measure the load on the server group. The gauge for measuring bandwidth uses the same probe used by the latency gauge for measuring the time it takes to receive a reply. An additional probe measures the size of the reply and calculates the bandwidth based on these values. A probe measuring the size of the request queue indicates whether a server group is overloaded.

3.3.2 Repair Execution

The final component of our adaptation framework is a translator that interprets repair scripts as operations on the actual system (Figure 1, item 5). The nature of these operations will depend heavily on the implementation platform. In general, a given architectural operation will be realized by some number of lower level system reconfiguration operations. Each such operator can raise exceptions to signal a failure. These are then propagated to the *Model Layer*.

To illustrate, the specific operators and queries supported by the runtime system in our example are listed in Table 1. These operators include low-level routines for creating new request queues, activating and deactivating servers, and moving client communications to a new queue. The operations at the *Model Layer*, describe in Section 3.2.1, are translated into calls on the operations in the *Runtime Layer* (Table 1) to effect the actual change in the system.

4 Implementation

Previously, the work on tools software architecture has mostly focused on design-time support. We have adapted these tools so that they can be used as runtime facilities. Specifically, AcmeStudio, an architecture design environment, can now make available an architectural description at runtime. This description can be analyzed by runtime versions of our Armani constraint checking and performance analysis tools, as well as be manipulated by our repair engine. Collectively, these tools implement the *Model Layer* elements in Figure 1.

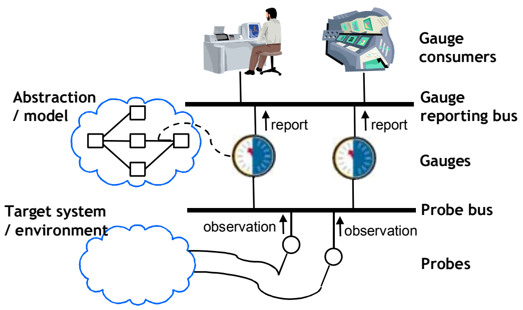


Fig. 4. Gauge Infrastructure

Table 1. Environment Manager Operators and Queries

CreateReqQueue()	Adds a logical request queue to *Req-queue* machine in Figure 2.
findServer([string cli_ip, float bw_thresh])	Finds a spare server that has at least *bw_thresh* bandwidth between it and the client.
moveClient(ReqQ newQ)	Moves a client to the new request queue.
connectServer(Server srv, ReqQ to)	Configures a server so that it pulls client requests out of the *to* request queue.
activateServer ()	Signals that the server should begin pull requests from the request queue.
DeactivateServer()	Signals that a server should stop pulling requests from the request queue.
remos_get_flow (string clIP, string svIP)	This is a Remos API call that returns the predicted bandwidth between two IP addresses.

In terms of monitoring, we have developed prototype probes for gathering information about networks, based on the Remos system [19]. Remos has two parts: (1) an API, that allows applications to issue queries about bandwidth and latency between groups of hosts; and (2) a set of *collectors* that gather information about different parts of the network [21]. A probe uses Remos to collect the information required for the probe and distributes it as events using the Siena wide area event bus [6]; gauges listen to this information and perform calculations and transformations to relate it to the software architecture of the system.

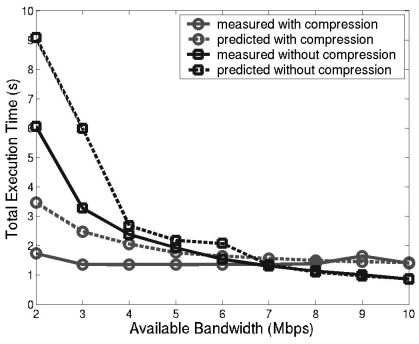
Currently, we have hand-tailored support for translating APIs in the *Model Layer* to ones in the *Runtime Layer* that need to be changed for each implementation. Our work in this area will concentrate on providing more general mechanisms where appropriate, and perhaps using off-the-shelf reconfiguration commands for commercial systems.

With respect to the *Task Layer*, we are actively investigating effective means for specifying user tasks, as part of our broader research in the Aura project at Carnegie Mellon University [27,33].

$$t_{\text{uncompressed}} = \frac{\text{size}}{available\_bw}$$

$$t_{\text{compressed}} = \frac{\text{size}}{comp\_speed} + \frac{\text{size} * comp\_ratio}{available\_bw}$$

(a)



(b)

Fig. 5. (a) The Analytical Compression Model; (b) Comparing Measured and Predicted Performance

For our modeling and analysis approach to be feasible, we need to have some confidence that the analysis at the *Model Layer* is relevant at the *Runtime Layer*. To explore specific data points of this, we have conducted some initial experimentation with comparing predicted performance and measured performance. Our simple experimental testbed was a client-server application where the repair tactic was to use compression to make more effective use of the available bandwidth [17]. In this case, we used simple analytical models instead of queuing models – shown in Figure 5(a).

The model variables in italics have to be determined at runtime. For our prototype (where compression uses gzip), the compression ratio (*comp_ratio*) depends on the data type (text, JPEG, etc.) and is simply determined through look up in a predefined table. The compression speed (*comp_speed*) is machine dependent. It is estimated based on benchmarks. Finally, the estimated network throughput (*available_bw*) is obtained using the Remos system.

Figure 5(b) shows the result of a set of experiments performed on a dedicated testbed. The testbed allows us to vary the available bandwidth (x-axis) by generating a variable competing UDP stream. The y-axis shows execution, both estimated (dashed lines) and measured (full lines). The experiments show two interesting results. First, the crossover point for the estimated execution time with and without compression happens at about the same point as the crossover point for the measured execution times with and without compression (indicated by the arrows in Figure 5(b)). This shows that the choice of tactics based on an analytical model will have the desired effect in the implementation. Second, around the crossover point, the execution times for the different tactics are very similar, suggesting that even if the client would pick the wrong tactic (for example because of a probe value with an unusually large error), the impact on performance would be minimal.

5 Related Work

Considerable research has been done in the area of dynamic adaptation at an implementation level. There are a multitude of programming languages and libraries that provide dynamic linking and binding mechanisms (e.g., [15,16]), as well as exception handling capabilities and distributed debugging [14]. Systems of this kind allow self-repair to be programmed on a per-system basis, but do not provide external, reusable mechanisms that can be added to systems in a disciplined manner, as with an architecture-driven approach.

There is a large body of research in the area of pervasive computing (e.g., [2]) and many companies are exploring support for this area. This research primarily focuses on user interface issues and the provision of low-level services and infrastructure in the environment. The notion of adaptation is hardwired into particular applications or services [5,9,18]. Again, our architecture-based approach provides a general solution that supports adaptation of applications and systems for which it is not explicitly supported.

The BBN QuO system [20] extends CORBA to support applications that adapt to resource availability. One aspect of the system is that users can define operating regions. The runtime system monitors the application and execution environment, and invokes application specific handlers when the application changes operating region. QuO is a specific example of an adaptive and reflective middleware system, which in general do not have an explicit architectural model of the application.

There has been some related research on architecture-based adaptation. However, this research relies on specific architectural styles, and implementations that match these styles [13,24]. In this paper, we have concentrated on how architectural models can be used to guide adaptation in a pervasive system, and the extensions need to software architectures to make them useful in a dynamic setting. In our broader approach we decouple the style from the system infrastructure so that developers have the flexibility to pair an appropriate style to a system based on its implementation and the system attributes that should drive adaptation. To accomplish this we have introduced some new mechanisms to allow "runtime" styles to be treated as a design parameter in the runtime adaptation infrastructure. Specifically, we have shown how styles can be used to detect problems and trigger repairs. We have also provided mechanisms that bridge the gap between an architectural model and an implementation – both for monitoring and for effecting system changes.

6 Conclusions and Future Work

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

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

We illustrated how the technique can be applied to performance-oriented adaptation in a pervasive computing environment with mobile users, time-varying resources, and heterogeneous devices.

For future research we need to be able to develop mechanisms that provide richer adaptability for executing systems. We also need new monitoring capabilities, and reusable infrastructure for relating monitored values to architectures. Finally, we need new analytical methods for architecture that will permit the specification of principled adaptation policies. Additionally we see a number of other key future research areas. First is the investigation of more intelligent repair policy mechanisms. For example, one might like a system to dynamically adjust its repair tactic selection policy so that it takes into consideration the history of tactic effectiveness: effective tactics would be favored over those that sometimes fail to produce system improvements. Second is the link between architectures and tasks. We need to further explore both how to specify user tasks and the precise interaction between them and the architectural parameters and constraints.

Acknowledgements

DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616, supports this work. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA.

References

1. Abowd, G., Allen,R., and Garlan, D. Using Style to Understand Descriptions of Software Architectures. In Proceedings of SIGSOFT'93: Foundations of Software Engineering, December 1993.
2. Abowd, G., Burmitt, B., and Shafer, S. (Eds). Ubicomp 2001: Ubiquitous Computing - Third International Conference Atlanta, Georgia, USA, September 30 - October 2, 2001 Proceedings. Lecture Notes in Computer Science **2201**, Springer, October 2001.
3. Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, June 1997.
4. Bertsekas, D. and Gallager, R. Data Networks, Second Edition. Prentice Hall, 1992. ISBN 0-13-200916-1.
5. Bollinger, J., and Gross, T. A Framework-Based Approach to the Development of Network-Aware Applications. *IEEE Transacations on Software Engineering (Special Issue on Mobility and Network Aware Computing)* **24**(5):367-390, May 1998.
6. Carzaniga, A., Rosenblum, D.S., and Wolf, A.L. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, July, 2000.
7. Clements, P., Bass, L., Kazman, R., Abowd, G. Predicting Software Quality by Architecture-Level Evaluation. In Proceedings of the Fifth International Conference on Software Quality, Austin, TX, October 1995.

8. Dashofy, E., van der Hoek, A., and Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 2001.
9. Flinn, J., Narayanan, D., Satyanarayanan, M. Self-Tuned Remote Execution for Pervasive Computing. In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Oberbayen, Germany, May 2001.
10. Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng, , New Orleans, LA, December 1994.
11. Garlan, D., Monroe, R.T., and Wile, D. Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems. Leavens, G.T., and Sitaraman, M. (eds). Cambridge University Press, 2000 pp. 47-68.
12. Garlan, D., Schmerl, B.R., and Chang, J. Using Gauges for Architecture-Based Monitoring and Adaptation. The Working Conference on Complex and Dynamic System Architecture. Brisbane, Australia, December 2001.
13. Gorlick, M.M., and Razouk, R.R. Using Weaves for Software Construction and Analysis. Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991.
14. Gorlick, M.M. Distributed Debugging on $5 a day. Proceedings of the California Software Symposium, University of California, Irvine, CA, 1997 pp. 31-39. Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. Proceedings of 5th European Software Engineering Conference (ESEC '95), Sitges, September 1995. Also published as Lecture Notes in Computer Science 989, (Springer-Verlag), 1995, pp. 137-153.
15. Gosling, J. and McGilton, H. The Java Language Environment: A White Paper. Sun Microsystems Computer Company, Mountain View, California, May 1996. Available at http://java.sun.com/docs/white/langenv/.
16. Ho, W.W. and Olsson, R.A. An Approach to Genuine Dynamic Linking. Software – Practice and Experience 21(4):375—390, 1991.
17. Hu, N. Network Aware Data Transmission with Compression. In Selected Papers from the Proceedings of the Fourth Student Symposium on Computer Systems (SOCS-4) Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-01-164, October 2001.
18. Krintz, C., and Calder, B. Reducing Delay with Dynamic Selection of Compression Formats. Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, California, USA, August 2001.
19. Lowekamp, B., Miller, N., Sutherland, D., Gross, T., Steenkiste, P., and Subhlok, J. A Resource Query Interface for Network-aware Applications. Cluster Computing, 2:139-151, Baltzer, 1999.
20. Loyall, J.P., Schantz, R.E., Zinky, J.A., and Bakken, D.E. Specifying and Measuring Quality of Service in Distributed Object Systems. In Proceedings of the 1st IEEE Symposium on Object-oriented Real-time Distributed Computing, Kyoto, Japan, April 1998.
21. Miller, N., and Steenkiste, P. Collecting Network Status Information for Network-Aware Applications. IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.

22. Monroe, R.T. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-98-163.
23. Moriconi, M. and Reimenschneider, R.A. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI International, March 1997.
24. Oreizy, P., Medvidovic, N., and Taylor, R.N. Architecture-Based Runtime Software Evolution in the Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, April 1998, pp. 11—15.
25. Oreizy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3):54-62, May/June 1999.
26. Perry, D.E., and Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* **17**(4):40-52, October 1992.
27. Satyanarayanan, M. Pervasive Computing: Vision and Challenges. IEEE Personal Communications, pp. 10-17, August 2001.
28. Shaw, M., and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
29. Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering, June, 1998.
30. Stafford, J., Richardson, D.J., and Wolf, A.L. Alladin: A Tool for Architecture-Level Dependence Analysis of Software. University of Colorado at Boulder, Technical Report CU-CS-858-98, April 1998.
31. Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6):390-406, 1996.
32. Vestel, S. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.
33. Wang, Z., and Garlan, D. Task-Driven Computing. Carnegie Mellon University School of Computer Science Technical Report CMU-CS-00-154, May 2000.