Software Architecture-based Adaptation for Grid Computing

Shang-Wen Cheng, David Garlan, Bradley Schmerl,
Peter Steenkiste, Ningning Hu

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{zensoul,garlan,schmerl,prs,hnn}@cs.cmu.edu

Abstract

Grid applications must increasingly self-adapt dynamically to changing environments. In most cases, adaptation has been implemented in an ad hoc fashion, on a per-application basis. This paper describes work which generalizes adaptation so that it can be used across applications by providing an adaptation framework. This framework uses a software architectural model of the system to analyze whether the application requires adaptation, and allows repairs to be written in the context of the architectural model and propagated to the running system. In this paper, we exemplify our framework by applying it to the domain of load-balancing a client-server system. We report on an experiment conducted using our framework, which illustrates that this approach maintains architectural requirements.

1. Motivation and Approach

Grid computing infrastructures [10,14] offer a wide range of distributed resources to applications. However, the heterogeneity of both the network and computing resources, and the dynamic load conditions make, adaptation an important requirement for grid applications. For example, the applications must be able to adapt themselves at runtime to handle such things as resource variability (network bandwidth, server availability, etc.) and system faults (servers and networks going down, failure of external components, etc.). If the system is not adaptive, it will often have unacceptably poor performance. In the past, managing and changing systems required human oversight, but in order to be practical, grid applications must be able to adapt automatically, with minimal human intervention.

Adaptability in most applications so far has been implemented in a fairly ad hoc fashion. The code that deals with adaptation is typically embedded in the application code. While this may work for local adaptation (i.e., for a single node or communication channel), it does not work well for global adaptation (e.g., that requires changes in the structure of the application) or in cases where multiple adaptation operations have to be coordinated. It also complicates both the application and adaptation code and makes the reuse of adaptation strategies impossible.

In this paper we present an alternative approach in which system models – in particular, software architectural models – are maintained at runtime and used as a basis for system reconfiguration and adaptation. 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 and connectors [20,22]. As a basis for self-repair, the use of architectural models has a number of nice properties: (1) An architectural model can provide a global perspective on the system allowing one to determine non-local changes to achieve some property; (2) architectural models can make "integrity" constraints explicit, helping to ensure the validity of any change; and (3) 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.

Our approach is based on the three-layer architecture shown in Figure 1:

- The *Runtime Layer* consists of the application itself, together with its operating environment (networks, processors, I/O devices, communications links, etc.) (1). It also includes a monitoring infrastructure that captures information that is relevant to the application. This information is filtered and presented to the model layer in architecture-relevant terms (2).
- The *Model Layer* is responsible for interpreting system observations. It consists of an architectural model of the system (3), together with an architecture manager (4) that determines whether a system's

Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)
1082-8907/02 $17.00 © 2002 IEEE

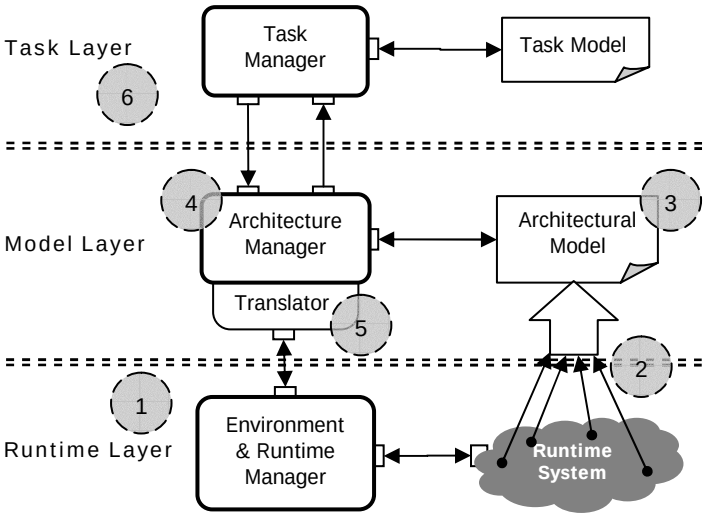

Figure 1 Adaptation Framework.

runtime behavior is within the envelope of acceptable ranges according to the architecture. The architecture manager verifies whether the application goals and constraints are satisfied, and if not, it can adapt the application using a repair handler. Repairs are propagated down to the running system (5).

- The *Task* Layer is responsible for setting overall system objectives (6). In this context, it can for example determine what applications should execute based on external policies. It can also set performance objectives and resource constraints for applications. These profiles will be used by the model-layer to guide adaptation. We will not discuss the task layer any further in this paper.

To illustrate how the approach works, consider a storage infrastructure consisting of a set of server groups that provide information to a set of users (Figure 3). Each server group consists of a set of replicated servers (Figure 2), 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. The goal is to use adaptation to maintain two inter-related system qualities. 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.

The remainder of this paper is organized as follows. We first present a short overview of software architectures. We then describe the main mechanisms underlying software architecture-based adaptation, and we discuss an experiment that illustrates the usefulness of our approach. We conclude with related work and a summary.

2. Software Architectures

The centerpiece of our approach is the use of architectural models. We use a simple scheme in which an architectural

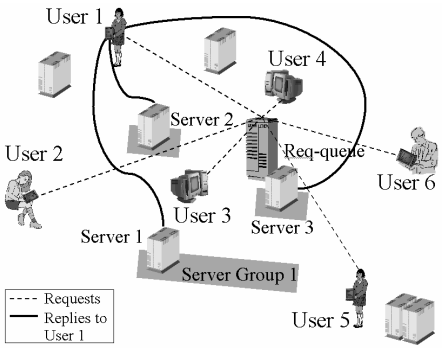

Figure 3. Deployment Architecture.

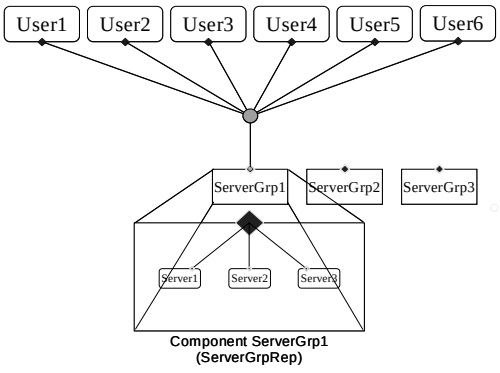

Figure 2. Software Architecture.

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 [18]. 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. In the software architecture illustrated in Figure 2, 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, elements in the graph can be annotated with a property list. For example, properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). The software architecture can also include a set of constraints that must be maintained. Constraints can, for example, specify that some property value must always be within a certain range. One of the advantages of architectural descriptions is that they provide opportunities for automatic verification of the constraints. In our design, the *Task Layer* specifies the performance profile for the application in the form of threshold constraints, e.g., *average latency* < *maxLatency*. These constraints can then be checked dynamically to see if the system is functioning within bounds.

3. Software Architecture-based Adaptation

The software architectures outlined above are design-time artifacts, and are created and analyzed using design-time tools. The idea behind software architecture-based adaptation is to make these models and analysis tools available at runtime so they can guide adaptation. This section discusses the main mechanisms: a *monitoring infrastructure* that provides information about the applications and the runtime environment, *repair strategies* that adapt the application, and *adaptation operations* that are used by the repair strategies to perform adaptation.

3.1. Monitoring Infrastructure

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 [16]) 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.

For instance, in the example above we must deploy a gauge that captures the *averageLatency* property of each client. Similarly, we want 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 information provided by these gauges will be used when making adaptation decisions.

3.2. 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 style-specific operators described in Section 3.3. 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

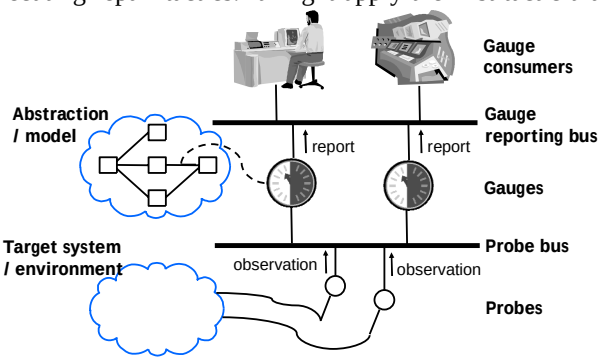

Figure 4. The Monitoring Infrastructure.

```
1  invariant r : averageLatency <= maxLatency
2  !→ fixLatency(r);
3
4  strategy fixLatency (badRole : ClientRoleT)={
5   let badClient : ClientT =
6     select one cli : ClientT in self.Components |
7       exists p : RequestT in cli.Ports |
8         attached(badRole, r);
9    if (fixServerLoad(badClient)) {
10    commit repair; }
11   else if (fixBandwidth(badClient,badRole) {
12    commit repair; }
13   else {abort ModelError;}
14 }
15
16 tactic fixServerLoad (client:ClientT) :boolean={
17  let loadedServerGroups :set{ServerGroupT}=
18   select sgrp:ServerGroupT in
19      self.Components |
20        connected(sgrp,client) and
21        sgrp.load > maxServerLoad;
22  if (size(loadedServerGroups) == 0)
23    return false;
24  foreach sGrp in loadedServerGroups {
25    sgrp.addServer(); }
26  return (size(loadedServerGroups)>0);
27
28  tactic  fixBandwidth(client:ClientT
29                       role:ClientRoleT):boolean={
30  if (role.bandwidth>=minBandwidth) {
31    return false;}
32  let oldSGrp: ServerGroupT =
33    select one sGrp:ServerGroupT in
34      self.Components |  connected (client,sGrp);
35  let goodSGrp : ServerGroupT =
36    findGoodSGrp(client,minBandwidth);
37  if (goodSGrp != nil) {
38    client.move (goodSGrp);
39    return true;
40  } else {
41    abort NoServerGroupFound;
42 }}
```

Figure 5. An Example Repair Strategy.

succeeds. Alternatively, it might sequence through all of the tactics.

Figure 5 illustrates the repair strategy and tactics associated with the latency threshold constraint in our example. Our analysis pointed us to the need for two different repairs if the observed latency rises above a threshold, as reported in [6]. The first is to add a new server to a server group if the server group is overloaded and the second is to move a client to a new server group that has better bandwidth to the client. A third repair (not shown) reduces the number of servers in a server group if the server group is underutilized.

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.

3.3. Architecture Adaptation Operators

Repair strategies use *architecture adaptation operators* to modify the architecture of the application. These operators will be specific to the structure of the architecture (this is called an *architecture style*). In the most generic case, architectures can provide primitive operators for adding and removing components and connections [6].

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, deleting the role currently connecting the client to the connector that connects it to a server group, and performing 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.

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.

Table 1. Environment Manager Operators and Queries.

The above operations 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.

The final component of our adaptation framework is a translator that interprets the actions of the repair scripts at the model layer as operations on the actual system at the runtime layer (Figure 1, item 5). The nature of these operations will depend heavily on the implementation platform. 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.

4. Framework Implementation

We have implemented a prototype implementation of the framework based on the set of Acme architectural design tools that we have developed previously [21]. A Java library, called AcmeLib, is used to parse Acme descriptions and check whether architectural constraints are satisfied. If constraints are violated, another component of our framework (Tailor) is invoked to conduct repairs. These repairs adapt the architectural model stored in AcmeLib so that the constraints are maintained.

Changes to architectural properties are received via the monitoring infrastructure. The monitoring infrastructure is implemented in Java, and uses the Siena wide area event bus to communicate messages over the distributed system. Gauges are implemented using our gauge library which implements a gauge protocol that we have defined for gauge creation, communication, and deletion. Probes in the implementation and environment use the Remos system [16] and a set of application-specific probes. The application-specific probes are implemented using AIDE [2], which preprocesses Java source code to facilitate the instrumentation of the code. The probes report when particular methods have been called, so that bandwidth, latency, and server load can be calculated by the gauges. These events are also reported to a Siena bus. Currently, we have hand-tailored support for translating APIs in the *Model Layer* to ones in the *Runtime Layer*. Also, the repairs included in Tailor are handwritten, using a form that could be generated from the repair strategies in Figure 5.

5. Experimental Evaluation

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

The implementation that we used for our experiment was based on the example presented in this paper – that of a client-server system using replicated server groups communicating over a distributed system. We used this example because the architectural style of the system is amenable to automatic performance analysis [23], the results of which we can use to guide the development of our repairs, as described in more detail in [6].

This system is implemented in Java and has a set of change operations corresponding to the operations in Table 1, that are called via RMI to change the system. The clients send requests to an entity that splits the requests into queues, corresponding to the client's server group. Servers in a group pull information from the appropriate queue, and send a reply. The size of the reply is indicated by the client request.

The requirements and assumptions that fed into our analysis are:

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

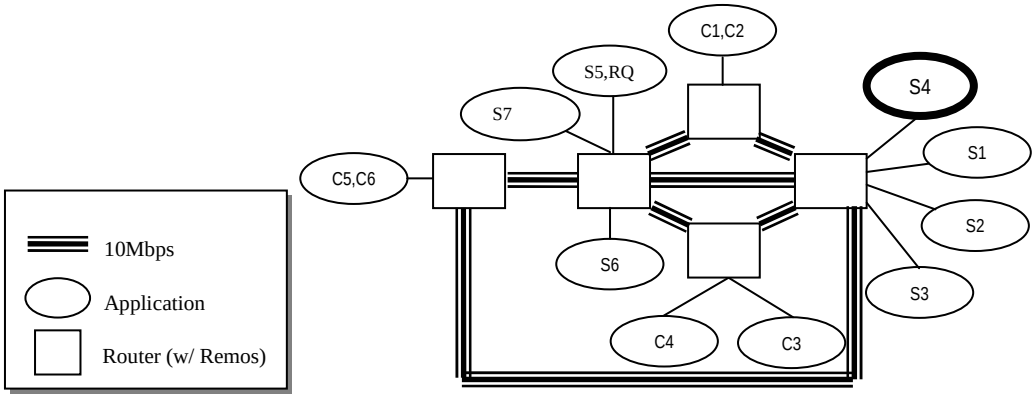

Figure 6. The Experimental Testbed.

- The average arrival rate of requests is approximately six per second.

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

5.1. Experimental Design

The experiment was conducted in a distributed setting inside a dedicated experimental testbed consisting of five routers and eleven machines (depicted in Figure 6), in which wed eployed the. Because we had access to fewer machines than processes, Clients 1 and 2 (C1 and C2 in the figure) share a machine, and the request queue shares a machine with Server 5 (S5). In the initial state, Servers 4 and 7 were spare servers that we could activate as repairs warranted. The routers are connected via 10Mbps links; each application node is connected to a router by a connection that is at least 10Mbps. The repair infrastructure was restricted to the machine running Server 4 (the thick ellipse in Figure 5), except for those parts of the infrastructure associated with monitoring and communication

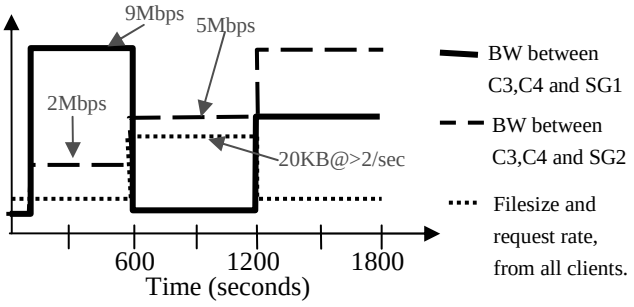

Figure 7. Bandwidth and Server Load Generation.

of observations, which were distributed throughout the environment.

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

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

The control and the experiment runs were executed under the conditions described above for a period of thirty minutes each. Figure 7 shows the stepping functions we used for generating bandwidth competition and server load. In the first two minutes, we ran the system in a quiescent state to give our gauges, probes, and system time to deploy and connect. In the following 8 minutes, we raised the bandwidth between the machines running Clients 3 and 4 (C3&4) and the machines representing Server Group 1 (SG1). In this period we would expect our repair strategies to migrate these clients to Server Group 2 (SG2). In the period 10 minutes to 20 minutes, we increased the server loads by increasing the file request size

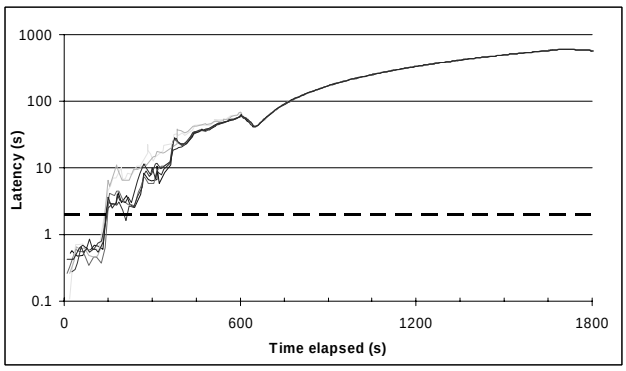

Figure 8. Average Latency for Control.

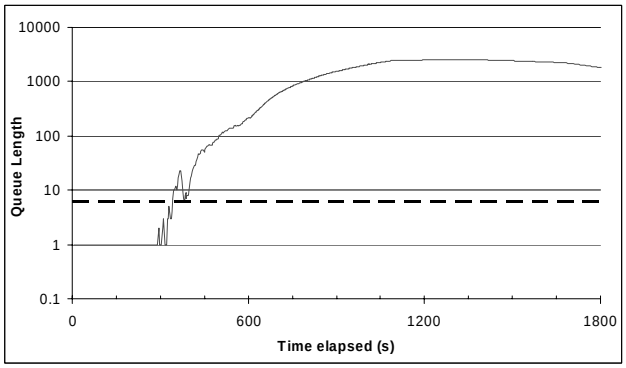

Figure 9. Server Load for Control.

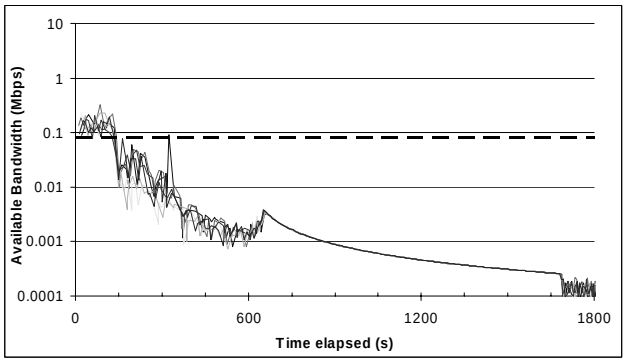

Figure 10. Available Bandwidth in Control.

and rate of messages sent from all clients (20KB, twice every second), while reducing the bandwidth to SG1. In the final 10 minutes, we increased the bandwidth between C3&4 and SG2. During the periods of high bandwidth between C3&4 and their respective server groups, we maintained moderate bandwidth (3Mbps) between the opposite server groups. We needed to restrict the competition in this way because of the limited resources on our testbed. In future work, we plan to run the experiments with more realistic bandwidth data, based on network traffic to Carnegie Mellon's web server.

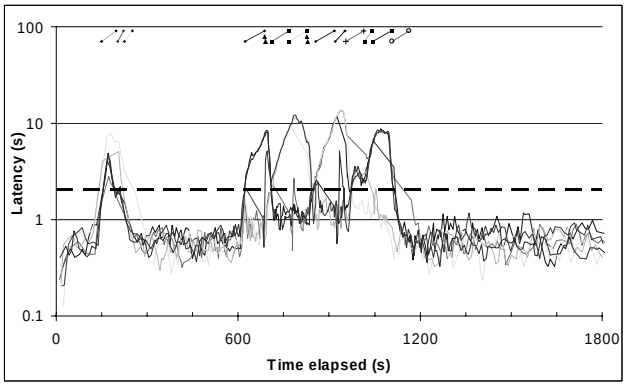

Figure 11. Average Latency under Repair.

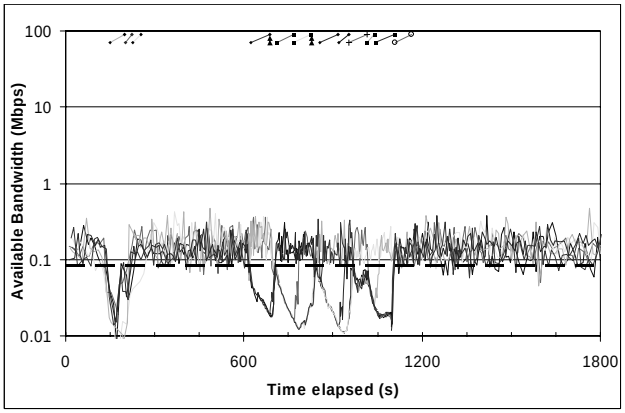

Figure 12. Available Bandwidth under Repair.

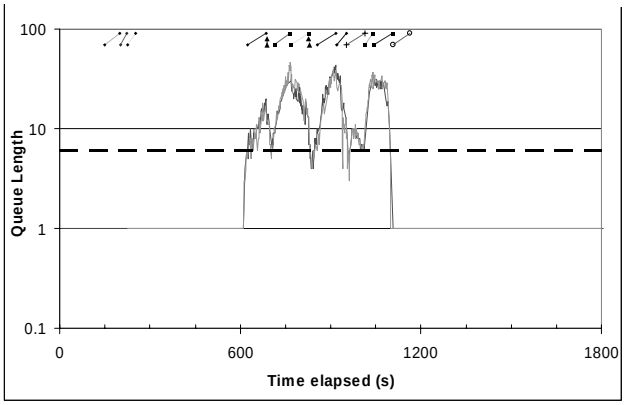

Figure 13. Server Load under Repair.

5.2. Results

The results for the control run (without adaptation) are shown in Figure 8 through Figure 10. The average latency, shown in Figure 8, continues to rise. Once the latency rises to above two seconds (at approximately 140 seconds for each client), it never falls below this required threshold. This is because the server load and bandwidth never

recover. In Figure 9, the server load increases dramatically as the experiment progresses. (Note that we measure server load by measuring the size of the queue of waiting client requests.) Similarly, the available bandwidth falls dramatically as the experiment progresses, as shown in Figure 10. The dashed line in both figures indicates the limits that we used to decide which repair tactic to execute. In Figure 9, a queue size of greater than six waiting requests indicated that the server was overloaded, and so the server repair should be tried. In Figure 10, an available bandwidth of less than 10Kbps indicated that there was not enough bandwidth. Note that for the control run, we overloaded the system so much that it never recovers. However, toward the end of our run the servers actually begin to recover.

Figure 11 through Figure 13 show the results obtained when our adaptation framework and repair strategies were applied under the same conditions as the control. Figure 11 shows a dramatic improvement in the average latencies experienced by the clients. Once our framework detects that client latency is above two seconds, a repair is invoked (either to move a client or add a server), and this improves the system performance as predicted by our design time analysis. In each of the figures for our experiment, the duration under which a repair is running is indicated by the lines at the top of the graph.[1] In fact, our framework has a positive effect on the available bandwidth because we are taking better advantage of different network links in our system after a repair. Our results for the server load show a marked improvement over the course of the experiment, except during the time that we increase the load on the server. During this time, we are continually performing repair. These repairs, encouragingly, do have a positive effect on the overall latency. Figure 13 shows the server load experienced during the run. Note that the only time that the server load rises above the constrained value is when we stress the servers.

5.3. Discussion

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

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

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

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

- The time that it takes to effect a repair averages 30 seconds. Most of this time is spent in communicating to create and delete gauges. Improving this time by caching gauges or relocating them (rather than destroying and creating new ones) should see our repair speed improve dramatically.
- The same network is being used to monitor the system as to run it. This means that when the available bandwidth is low, communication over our monitoring system is correspondingly slow. This produces a lag in the time when the bandwidth actually rises and the time it is noticed and repaired by our system. One way to address this is to use network Quality of Service (QoS) techniques to prioritize monitoring traffic.
- It is important to understand the underlying probe technology. The first Remos query for information about bandwidth between two nodes on the network takes several minutes because Remos needs to collect and analyze data. After this initial delay, the query is quite fast. To reduce this effect, we prequeried Remos so that subsequent queries were much faster. Again, this reduced the time of our repairs. In general, this points to the need for more sophisticated probe technologies that need to be provided for caching or prefetching this information.
- In some instances, the effects of a repair on a system will take time. For example, adding a new server to a server group will not immediately reduce the overall load on the server group. Without taking this effect into account, unnecessary repairs are likely to occur (for example, to continue adding servers or to move clients). This type of delay is something that can only be gleaned from experience of running the repairs, and points to the need for a more sophisticated repair engine that can monitor repairs and their effects, and use this to adapt its repair policy.

Although we do not expect our approach to compete with hand-tailored, per-application adaptation, we believe that this approach will save time in engineering adaptation into applications that require it but do not possess it, in

analyzing those repairs, and in changing them as required. However, this would be moot if the repairs did not improve the situation. These results show that we do get improvement by applying our framework – how this improvement compares to hand-tailored adaptation is an area of future work.

6. Related Work

There has been a lot of research on the development of adaptive applications, e.g., [9,15]. However, in most cases the adaptation process is hardwired into the application code, making it difficult to characterize or modify the adaptation rules. However, the need for a more systematic approach has been recognized. For example, [1] describes a generic adaptation framework for point-point streaming applications, and [4] proposes an architecture that separates application code, although only for local adaptation. Our architecture-based approach provides a more general solution that supports adaptation of applications and systems for which it is not explicitly supported.

The BBN QuO system [17] extends CORBA to support applications that adapt to resource availability. In this system users can define operating regions, while contracts specify the performance expectations for application modules. 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, but it does not have an explicit architectural model of the entire application. A similar approach is outlined in [25].

Given that adaptation relies on access to accurate system information, several monitoring tools in support of network-aware applications have been developed [26,27]. Like Remos, these tools can play the role of probes in our architecture. JAMM [24] is an environment that gives users access to wide range of information on the distributed system. This design is similar to our proposed gauge infrastructure. For example, both gauges and JAMM give users access to information on both the runtime environment and the application, making it possible to not only detect problems, but also diagnose them. A grid information services architecture that has been implemented as MDS-2 in Globus is described in [7] . While many details are different, its high-level architecture roughly matches the probe/gauge component of our architecture. It distinguishes between information providers (similar to our probes) and aggregate directory services (similar to gauges).

There has been some related research on architecture-based adaptation. However, it relies on specific architectural styles, and implementations that match these styles [13,19]. In this paper, we have described mechanisms that bridge the gap between an architectural model and an implementation – both for monitoring and for effecting system changes.

7. Conclusion and Future Work

In this paper we presented a technique for using software architectural models to automate adaptation in systems. This approach has a number of 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.

We described an experiment that illustrates that our approach can maintain architectural constraints. The experiment also directs us to future work. For example, we plan to investigate strategies for selecting which repairs are best in given circumstances. Our experiment simply chose to repair the first client that reported an error, and the tactic chosen was to prioritize server load repairs. Smarter approaches include fixing the client that is experiencing the worst latency first, choosing the tactic that contributes the most to the latency, and other adapting schemes alluded to in Section 5. Similarly, we need to be able to capture in a repair strategy the desired behavior if no repair will improve the situation – for example, if the server load is too high and there are no available servers to add to a server group. In such a case it may be necessary to alert a human observer for manual intervention.

Another area of future research is to provide additional tool support for specifying repairs and the mappings between the architectural level and the runtime level. Currently, we have hand-coded solutions for these components of our framework. We are using the example described in this paper, and others, to elicit the requirements for such components, in order to make the engineering of these components easier.

Acknowledgements

This work is supported in part by DARPA under Grants N66001-99-2-8918 and F30602-00-2-0616. 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] 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.

[2] Calnan, P. Semantic-based Code Transformation. MS Thesis Proposal, Department of Computer Science, Worchester Polytechnic Institute, Massachussets, March 2002.

[3] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC2000), Portland OR, July, 2000.

[4] Chang, F. and Karamchety, V. Automatic Configuration and Run-time Adaptation of Distributed Applications. Proc. the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00), pages 11-20, August 2000.

[5] Cheng, S-W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N. Software Architecture-based Adaptation for Pervasive Systems. International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing, April 8-11, 2002.

[6] Cheng, S-W., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P. Using Architectural Style as the Basis for Self-repair. The Working IEEE/IFIP Conference on Software Architecture 2002, Montreal, August 25-31, 2002. To appear

[7] Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C. Grid Information Services for Distributed Resource Sharing. Proc. the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[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] Foster. I. and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. I. Foster, C. Intl J. Supercomputer Applications, 11(2):115-128, 1997.

[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] Grimshaw, A. and Wulf, W. The Legion Vision of a Worldwide Virtual Computer. Communications of the ACM, Jan 1997, Vol 40, No 1.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Perry, D.E., and Wolf, A. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4):40-52, October 1992.

[21] Schmerl, B., and Garlan, D. Exploiting Architectural Design Knowledge to Support Self-repairing Systems. Proc. the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE2002), July 2002.

[22] Shaw, M., and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[23] Spitznagel, B. and Garlan, D. Architecture-Based Performance Analysis. Proc. the 1998 Conference on Software Engineering and Knowledge Engineering, June, 1998.

[24] Tierney, B., Crowley, B, Gunter, D., Holding, M., Lee, J. and Thompson, M. A Monitoring Sensor Management System for Grid Environments. Proc. the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00), pp. 97-104, August 2000.

[25] Vraalsen, F, Advt, R., Mendes, C., and Reed, D. Performance Contracts: Predicting and Monitoring Grid Application Behavior. Presented at GRID 2000 workshop, November 12, 2001, Denver.

[26] Wolski, R. Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. Proc. the 6th High-Performance Distributed Computing Conference (HPDC97), August 1997, pp. 316-325.

[27] Wolski, R. and Jayes, J. Predicting the {CPU} Availability of Time-shared Unix Systems. Proc. the Eighth IEEE Symposium on High Performance Distributed Computing {HPDC99}, August 1997, pages 105-112..