To appear in Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4), June 12-15, 2004

An Architecture for Coordinating Multiple Self-Management Systems

Shang-Wen Cheng An-Cheng Huang David Garlan Bradley Schmerl Peter Steenkiste
School of Computer Science, Carnegie Mellon University
{zensoul, pach, garlan, schmerl, prs}@cs.cmu.edu

Abstract

A common approach to adding self-management capabilities to a system is to provide one or more external control modules, whose responsibility is to monitor system behavior, and adapt the system at run time to achieve various goals (configure the system, improve performance, recover from faults, etc.). An important problem arises when there is more than one such self-management module: how can one make sure that they are composed to provide consistent and complementary benefits? In this paper we describe a solution that introduces a self-management coordination architecture and infrastructure to support such composition. We focus on the problem of coordinating self-configuring and self-healing capabilities, particularly with respect to global configuration and incremental repair. We illustrate the approach in the context of a self-managing video teleconference system that composes two pre-existing adaptation modules to achieve synergistic benefits of both.

1 Introduction

Software-based systems today increasingly operate in changing environments with variable user needs, resulting in a continued rise of administrative overhead for managing these systems. Thus, systems are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. A common approach to adding self-management capabilities to a system is to provide one or more external control modules, whose responsibility is to monitor system behavior, and adapt the system at run time to achieve various goals.

Self-management entails many different aspects, resulting in distinct dimensions of control. For instance, IBM's autonomic computing initiative views self-managing systems as typically exhibiting a subset of four capabilities: *self-configuring* (adapt automatically to dynamically changing environments), *self-healing* (discover, diagnose, and react to disruptions), *self-optimizing* (monitor and tune resources automatically), and *self-protecting* (anticipate, detect, identify, and protect themselves from any attacks) [9]. As another important dimension, system adaptation typically spans two scopes: large-scale global configuration, and incremental repair in response to local failures. Finally, different quality dimensions such as performance, security, and reliability may require management as well.

For many application domains, managing a system requires managing multiple dimensions. For example, consider a video conferencing system with different user applications in a heterogeneous network environment, where the aim is to provide the best service at the lowest cost. At once, several potential dimensions of control exist, including composition, change, performance, and cost of service, each corresponding to different domain expertise, and thus, modules of control. Many self-management modules (SMs) are available today, each typically capable of addressing a distinct aspect of self-management. The challenge is to allow developers to coordinate multiple distinct management modules together in a coherent and consistent fashion to manage a system.

A simple approach to do this is to let the different SMs run independently. This can potentially lead to conflict, inconsistencies, and modules working at cross purposes. Another approach is to reimplement all of the combined self-management capabilities into one monolithic control module, with the obvious problem that it is ad hoc, not cost-effective, prevents reuse, does not scale, and results in an overly complex system.

A better approach is to combine multiple SMs in a coordinated fashion. However, such an approach has the difficult challenges of maintaining consistency in information acquisition and internal models and ensuring coherent decisions. In this paper, we propose a coordination architecture embodying points of shared system access, model translation, and decision control patterns to integrate multiple SMs. We will highlight the challenges, describe our approach, and present a case study to demonstrate the approach.

2 Related Work

Recently, considerable research has been done on self-managing systems, including work from IBM's autonomic computing initiative. An important challenge in their work

is the coordination of multiple autonomic elements to form a cooperative system, where an autonomic element embodies a system element managed by an autonomic manager, similar to what we term *self-management module*. Our work shares a similar challenge in that we attempt to coordinate multiple SMs to manage one system, which itself contains multiple elements.

To the best of our knowledge, no one has explicitly addressed the problem of coordinating multiple SMs to manage a single system. A few approaches (e.g., [22]) show promise of a comprehensive architecture to address multiple aspects of self-management, but have not carried the results through to implementation. In comparison, our work addresses this problem, with a focus on the aspects of global configuration and incremental adaptation in the context of self-configuring and self-healing capabilities. Various researches address the dimensions of self-management mentioned before, including the areas of system composition, smart components, and incremental adaptations.

Previous work on service composition frameworks have attempted to automate the generation of global system configuration given certain constraints and/or optimization criteria. Most of these efforts explore a path-based (e.g., Panda [28] and Ninja [13]) or graph-based (e.g., SWORD [27]) service composition model to transform the given input(s) to the desired output using a series of format adaptors. The Libra framework [19] aims to automate the optimal composition of services across the wide-area network using service-specific knowledge.

Research on smart components (e.g., smart servers, smart databases) that adapt to changes in the environment provide building blocks for self-management systems [21, 23]. Some of these have the ability to form a self-organizing system (e.g., [12]). Finally, several researchers take the approach of using externalized mechanisms to dynamically adapt a running system [5, 7, 8, 14, 26]. Due to the nature of dynamic, run-time adaptation, their work have focused on incremental adaptation.

Depending on application domains, different self-management decisions might be based on different quality attributes such as performance, security, and reliability. In more complex cases, a combination of attributes may need to be considered by using utility models. For example, some of the work on dynamic adaptation applied their SMs to systems with primarily a performance concern [5, 7, 14].

3 Coordinating Self-Management Modules

To harness multiple self-management capabilities for a system, we propose to use more than one existing SM in a coordinated fashion. However, the question is, how can one make sure that the modules work together to provide consistent and complementary benefits?

Adopting IBM's autonomic computing control loop point of view, we can identify three phases of interaction:

Sense The SM first senses and aggregates information from the system and updates its model(s) of the system.

Evaluate Sensed information is analyzed, based on certain metrics, to decide on the course of corrective action.

Act The planned course of action is carried out on the system to improve or correct the state of the system.

Central to all three of these are internal models to make the overall control work. When multiple SMs have to be coordinated, each of these phases introduces a potential point of conflict and inconsistency. As a result, there are three technical challenges to address.

(1) Consistent system access. Each SM needs to obtain some information of the managed system for decision-making. Each SM also needs to change the managed system. For instance, SMs for a video conferencing system might need to know the connection latency or component cost. If the the same information is collected from multiple, different sources—e.g., one source reports a cost of $10, and another a cost of $100—the SMs may arrive at different conclusions about the system, and thus potentially make conflicting decisions. A different issue arises when a single observation such as latency may require multiple sources of information. In such a case, consistent interpretation in the sensors themselves is important. Likewise, when two SMs want to change the same components, their actions may need to be synchronized to ensure consistent outcome. Thus we need to ensure the consistency of sensed information from the environment and of actions on the system.

(2) Non-conflicting decision. The SMs must evaluate or interpret the sensed information to make decisions. To do so each SM needs to base its interpretation and evaluation on certain metrics, possibly in the form of utility models, to make decisions. These metrics might include performance, reliability, security, or cost, and, as mentioned before, the SMs might focus on different metrics. In the video conferencing example, a composition SM might evaluate metrics of overall service requirement, while a service-change SM might evaluate metrics on adding a user to the conference. Because of the different focus, the SMs can potentially make conflicting decisions. For example, given the same component cost of $100, the composition SM might consider it a low cost, while the service-change SM might consider it high. In a different scenario, the service-change SM might need to join a new user to the video conference, while the composition SM might instead want to recompose the entire conferencing session. If each works independently, the resulting changes could conflict with one another, and possibly leave the system in a broken state. Therefore, we need to ensure non-conflicting and complementary decisions.

(3) Consistent model. Each SM has an internal model of the system, which will most likely focus on different aspects and reveal different level of details of the system. For example, the service-change SM might maintain an architectural model of the video conferencing system, while the composition SM might keep a detailed structure of all of the conference elements. Because the models are the basis of self-management decisions, the coordinated SMs must have a consistent view to achieve synergy. Consequently, we need to ensure consistency across the models.

A solution should address all parts of the coordination problem by ensuring consistency in the sensed information and action, by coordinating metrics and decision, and by ensuring model consistency. In this paper, as a first step, we focus on addressing the first and third challenges. We briefly discuss the issues of decision coordination, but reserve a comprehensive solution of the second challenge for future work.

4 Our Approach

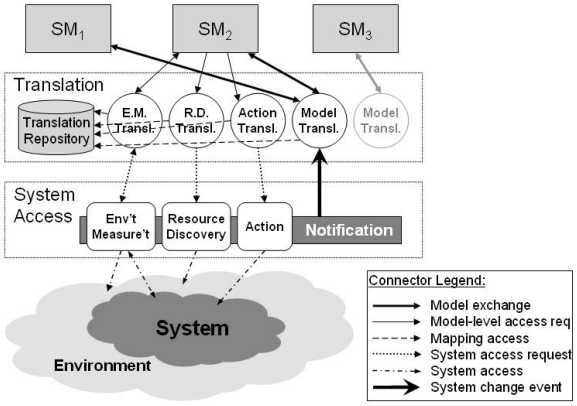

Figure 1. Coordination Architecture

Our approach to address the core issues of consistency and coherence identified in the previous section can be summarized in three conceptual parts. First, we identify three commonly recurring mechanisms for system access, and propose an infrastructure that shares these mechanisms to eliminate redundancy and conflicts due to system changes, and to ensure that all models across the various SMs reflect the system changes.

Second, in order for the SMs to cooperate on a decision, they need to exchange model information. Furthermore, the sharing of the system access requires a common representation of system information, which may differ from the SMs' internal models. Hence, we propose a translation infrastructure to enable model exchange and system access.

Third, in order to reach a single, coherent decision among multiple SMs, coordination of the decision process is crucial. Therefore, we propose to coordinate evaluation metrics among the SMs and to enforce a control pattern that allows the SMs to cooperatively make decisions.

These three results are embodied in a coordination architecture, shown in Figure 1. Self-management module 1 (SM_1) interacts with SM_2 via the model translator in the translation infrastructure. SM_3 corresponds to the general case. Each SM_i accesses the system and its surrounding environment using the system access infrastructure, also by way of translation. The system access components send notifications to the SMs via translation. Finally, each of the translation components uses the translation repository. In the following subsections, we describe in detail the roles of each part, and how they work together.

This approach provides common infrastructures to coordinate SMs and produce consistent and coherent self-management systems. The common infrastructure further hides lower-level system access details from the SMs, allowing developers of self-management modules to concentrate on the more abstract management logics. Our approach thus reuses common mechanisms, reduces the cost of composing multiple SMs, and potentially promotes the development of more SMs for others to reuse and compose self-management systems down the road.

4.1 System Access

Most SMs use these three mechanisms to get information into and out of the managed system:

Environment measurement. This mechanism supports the observation and measurement of various states of the system and the system environment, including component properties such as load and liveness, and connection properties such as latency and bandwidth. The sensing mechanism supports two ways to acquire environment information—monitoring, where information is pushed from the system, and querying, where information is pulled from the system

Resource discovery. This mechanism facilitates the discovery of available resources in the environment that are not part of the existing system, based on resource type and other criteria. For example, an SM for a video conferencing system might need to discover a new conferencing gateway to replace a failed gateway. Furthermore, the discovery might be based on proximity to existing users, load requirement, and cost.

Action component. This mechanism enables the SM to modify the configuration of the system. In the above example, after the discovery of the replacement gateway, the SM might use the action component to remove the old gateway from the system and put the replacement gateway into service with certain configuration settings.

Because of the importance and recurrence of these three mechanisms across SMs, when coordinating multiple SMs, we can eliminate redundancy by sharing these mechanisms among the SMs. Furthermore, since system access potentially changes the system, all models need to reflect the changes. Thus, our approach provides a shared system access infrastructure responsible for keeping the models in the various SMs updated through a notification mechanism when any one of the SMs accesses the system. This helps address the third challenge to ensure consistent models. Since not all changes affect all models, only the SMs whose models are affected need to be notified.

Consider an example where SM_2 removes an existing element from the system using the action component. The action component would notify the other SMs of this new change to the system. Next, SM_2 might want to find a replacement element, which it does through resource discovery. The resource discovery component would notify the other SMs about the newly discovered resources. As a third example, SM_2 might want to know some property of one of the new resources. It queries this property through the environment measurement component, which would then inform the other SMs of the queried property.

This shared system access infrastructure addresses the first challenge to ensure the consistency of sensed information from the environment, since all the SMs obtain their information from a single source. In addition, sharing environment measurement and resource discovery has the benefit of enabling performance optimization such as measurement caching. Sharing the action component facilitates synchronization of system changes to ensure consistent outcome in the system. In contrast, synchronizing multiple action components requires more complex organization scheme, protocols, and algorithms.

4.2 Translation Infrastructure

The second part of our coordination architecture is the translation infrastructure to enable the shared system access, which requires a common representation of system information, and the exchange of model information. To fulfill the two purposes, the translation infrastructure consists of separate translator components for exchange of model information and for each of the three system access functions. All of the translator components share a translation repository to maintain the necessary mapping information. Together with the notification mechanisms of the system access infrastructure, the translation infrastructure addresses the third challenge to ensure consistent models.

Note that there are potentially many different kinds of models that might need translation. However, we are most interested in models that reflect a run-time architecture of the system, such as a component-connector view. Our design of the translation infrastructure, in particular the different kinds of mapping knowledge, is thus based on this assumption.

Translation knowledge. The translation repository stores four kinds of mappings used by the translator components, namely type, element, operation, and error mappings. The use of a repository enables the translators to share the mapping knowledge.

Type mapping. The "type" refers to the class or category of an element and defines a set of properties that the element can have. The type mapping between two types t_1 and t_2 consists of a simple relation of their names, plus the relations of each of the properties of t_1 to the corresponding property of t_2. Each property is represented as a pair of property type and property name. Type mapping must exist between different models, and between each model and the system. For example, in a self-managed video conferencing system with two coordinated SMs, we might see the following type mapping for a gateway element.

	SM_1	SM_2	*Sys*
type	GatewayT	GW_Type	ServiceGW
prop1	N/A	(String, location)	(InetAddr, ip)
prop2	(float, cost)	(float, cost)	(float, cost)

This table shows a mapping of a single type among SM_1, SM_2, and the system. The mapping provides a relation of the type names *GatewayT*, *GW_Type*, and *ServiceGW*, and a set of relations between the properties of the types. Notice that not all properties have a correspondence in the other models and the system (e.g., location). In addition, some properties, like *location* and *ip* require transformation.

Element mapping. The "element" refers to an entity in a model or the system, and is a tuple of the entity's type and its properties. Given an element in a particular model, the element mapping allows us to obtain the corresponding element in another model, or the system. Thus, the element mapping is a simple relation between two elements. Continuing the previous example, we might see the following element map for a gateway element.

	SM_1	SM_2	*Sys*
(type, props..)	(GatewayT, cost=1.3)	(GW_Type, location="PA", cost=1.3)	(ServiceGW, ip=10.1.2.3, cost=1.3)

Operation mapping. The "operation" refers to a unit of action that the SM can issue to the action component, which then carries it out on the system. Unlike the type and element mappings, operation mapping only exists between each of the SMs and the system. The operation mapping between two operations op_1 and op_2 consists of a simple relation of the operation names, plus the relations of each of the parameters of op_1 to the corresponding parameters

of op_2. Each parameter is represented as a pair of parameter type and parameter name. For example, in the same self-managed video conferencing system, we might see the following operation map from SM_1 to the system:

	SM_1	*Sys*
op	start	start
param1	(GatewayT, src)	(ServiceGW, src)
param2	(float, timeout)	(float, timeout)

This table shows a mapping of an operation for starting a gateway element, in which the gateway parameter needs to be translated using the element mapping, but the timeout parameter does not. Notice, however, that a more generic parameter type can be specified to allow the operation to accommodate more cases.

Error mapping. The "error" refers to a problem that occurred during an operation. Like operation mapping, error mapping only exists between the system and each of the SMs. The error mapping between two errors ex_1 and ex_2 consists of a simple relation of the error identifiers, plus possible relations of the sources of error of ex_1 to the corresponding sources of ex_2. For example, consider the same video conferencing system, where we might have the following error map from the system to SM_1:

	SM_1	*Sys*
error	GatewayNotFound	GatewayHostNotFoundEx

Among the four types of mappings above, the type, operation, and error mappings are provided *a priori* to the translation infrastructure, and populated in the translation repository, before the system is deployed.

Types of translators. As mentioned before, there are four kinds of translator components. The *model translator* enables SMs to exchange model information with each other and to maintain consistency with the system. It uses the type mapping to transform individual elements of the source model to elements in the target model, generating element mappings in the process. For example, consider translating SM_2's model, which contains a gateway element (g_2) as shown in element mapping table, to SM_1's model. The model translator would search for `GW_Type` in the type mapping, find the target type `GatewayT`, create the target element (g_1) of that type, then use the property mapping to fill in the properties. Finally, it stores the resulting element mapping $(g_1 \leftrightarrow g_2)$ in the repository.

The system access translators work similarly to the model translator to enable the SMs to communicate with the system access infrastructure. When an SM interacts with the system, it refers to system entities and types using the vocabulary of its internal model. It is the responsibility of these translators to transform those references to specific system entities that the system access infrastructure can understand and manipulate. Responses from the system access infrastructure also needs to be translated back into the vocabulary of the accessing module.

The *environment measurement translator* uses the element mapping to translate the elements in queries from the SMs. The *resource discovery translator* uses the type mapping to translate the resource type of the request, and the element mapping to translate the discovered resource elements. The *action translator* uses the operation mapping to translate the actions specified by the SMs, then the element mapping to translate the individual parameters as necessary, and finally the error mapping to translate any resulting error.

4.3 Decision Coordination

The third part of our coordination architecture is the coordination of the decision process as the SMs use sensed information provided by the system access infrastructure to evaluate and act on a solution. Although we are currently still researching this problem, we believe that decision coordination can be achieved through coordinating evaluation metrics and enforcing a control pattern.

Evaluation depends on application-specific metrics, which may differ for different SMs. The problem is that the various metrics may be quite independent, or even conflicting, for example performance versus security. In such cases, the prioritization of the metrics is mostly a policy issue, so cannot be derived automatically. Consequently, our approach defers the coordination of metrics to the domain experts who are integrating the SMs. The domain experts must provide a compatible set of utility criteria to our coordination architecture to ensure that the SMs apply compatible metrics.

Compatible metrics for the SMs is only the first step, the next step is to ensure that when a problem arises, the SMs can cooperatively arrive at a coherent decision through some protocol of negotiation. In other words a control pattern must be established, and a few examples include:

Single-active Only one active at any time, with explicit yielding of control.

Balance of power Any module can veto the others.

Master-slave The master assigns tasks to the slave.

Agent-based Agent-like, peer-to-peer negotiations.

Democracy Modules use different metrics to decide on a solution, then majority voting determines an outcome.

One important factor that affects the choice of control pattern is the number of SMs coordinated. For any control pattern, the complexity of the interaction grows with the number of SMs coordinated. The more complex control patterns would be less practical for a large number of SMs. Relative priority of the self-management dimensions is another factor. Ultimately, the application domain and the stakeholders determine the best control pattern to enforce.

In summary, we expect to address the second challenge to ensure non-conflicting and complementary decisions by (1) determining a set of compatible evaluation metrics for all the SMs based on domain-specific knowledge, and (2) establishing a control pattern suitable to the application domain that will ensure a single coherent decision.

5 Case Study

To evaluate our approach, we performed a case study where we coordinated two existing SMs—Libra and Rainbow—to manage a single system. To test the resulting self-management coordination, we chose a video conferencing system as an example target system. The video conferencing system used did not dynamically adapt. It was therefore a good candidate for applying multiple dynamic adaptation techniques without being concerned with interfering adaptations from the system.

5.1 Overview

The Libra framework. The goal of the Libra framework is to dynamically compose a "service instance" (consisting of various components) that is optimized for the requirements and preferences specified in a particular user request, taking into consideration the global environment characteristics at the time of the request. In other words, Libra aims to provide the global configuration capability. To achieve this efficiently, Libra separates the domain-specific *knowledge* of composition from the generic *actions* involved in the actual composition. A service provider that wants to use the Libra framework to provide a service would translate its domain-specific knowledge into a "service recipe" that specifies what components are needed given certain requirements, what environment information should be obtained for optimization, and so on. The central element in Libra is the *synthesizer*, which interprets the recipe given by the provider and carries out the actions accordingly, e.g., finding the necessary components, querying for the needed environment information, calculating the optimal composition, and starting and connecting the components [19].

The Rainbow framework. Rainbow is an architecture-based, dynamic self-adaptation framework that monitors and incrementally adapts a target running system using the system's architectural model. The architectural model externalizes the reasoning of system properties and conditions for adaptation. The architectural model satisfies an architectural style, which defines a family of architectures with 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. The framework's leverage of architecture style enables analysis, system evolution, and reuse of both adaptation expertise and infrastructure, thus achieving cost-effectiveness [5, 10].

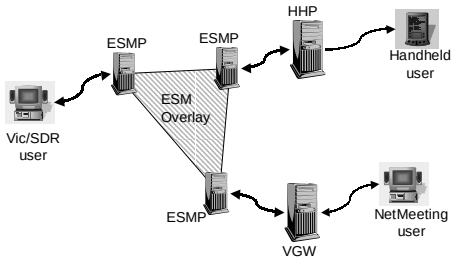

Figure 2. A video conferencing session example

Video conferencing system. Our target system is a video conferencing system that supports users with different conferencing applications and hardware capabilities and involves various components across heterogeneous network environments. Specifically, three types of user are supported: (1) Vic/SDR users, who use a combination of conferencing tools [2] designed for the MBone multicast testbed [1], (2) NetMeeting users, who use the NetMeeting application [24], and (3) Handheld users, who use a handheld device with an application that only receives video from a conferencing session.

To support these different users in a single video conferencing session, a number of service components are needed. For example, Figure 2 shows a video conferencing scenario that involves one Vic/SDR user, one NetMeeting user, and a handheld user. First, since Vic/SDR uses the Session Initiation Protocol (SIP) [16] for session setup while NetMeeting uses the H.323 protocol [20], a "video conferencing gateway" (VGW) must be used to translate the different negotiation protocols. Secondly, the handheld application does not have the capacity to perform the session negotiations. Therefore, a "handheld proxy" (HHP) is used to negotiate on behalf of the handheld user. Finally, the communication among the Vic/SDR user, the VGW, and the HHP requires IP multicast, which is not available in the wide-area network. Therefore, an "end system multicast" (ESM) approach is used that uses a number of "ESM proxies" (ESMPs) to provide the IP multicast functionality across the wide-area network.

5.2 Coordinating Libra and Rainbow

The Libra self-management framework has the expertise in global configuration, while Rainbow has strength in incremental adaptations. For managing a system, these two capabilities serve complementary roles, so it is natural to consider a synergistic composition of the two frameworks. Taking the approach outlined in Section 4, we coordinated the Libra and Rainbow frameworks by sharing a system access infrastructure, creating a translation infrastructure, and establishing a control pattern to coordinate decisions.

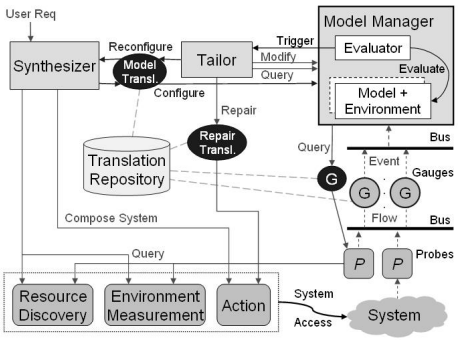

Figure 3. Rainbow-Libra Coordination

The resulting framework is shown in Figure 3, using matching shapes with Figure 1 to help relate corresponding parts. Synthesizer is one of the SMs. Tailor with the model manager together form the other SM. Details on the rest of the framework follow.

System access. As described earlier, the system access infrastructure provides the mechanisms of environment measurement, resource discovery, and action components, which are needed by the SMs, Libra and Rainbow. Our general approach to providing these capabilities has been outlined in Section 4.1. One part that is specific to this particular case study is the use of gauges and probes to support environment measurement.

Environment measurement involves two aspects: monitoring and querying. In other words, the gauges and probes support both push-based and pull-based methods for acquiring environment information. A probe is able to observe or measure certain properties of system elements. In the monitoring mode, the information gathered by a probe is periodically published to interested gauges, which then report the information to an SM.

On the other hand, when an SM wants to acquire the up-to-date value of a property, it can query the appropriate gauge, which asks the corresponding probe to get the current value of the property. The distinction between probes and gauges is that gauges serve the function of the translator for environment measurement.

Translation infrastructure. The translation infrastructure bridges the gap between the models of the SMs, and between the SMs and the system. In this case study, the Rainbow framework operates at the architecture level, so translation between the Rainbow module and the system is necessary. For system access, translation is realized via gauges for the environment measurement and resource discovery components, and a repair translator for the action component. The model translator works partly as we described, to translate between the system and Rainbow's model. Based on our analysis of the video conferencing system, we generated the initial type mappings of system element types to architectural element types for the translation infrastructure.

On the other hand, the Libra framework actually operates at the system level, so translation between the Libra module and the system is unnecessary. In addition, the translation between the Rainbow module and the Libra module only requires the same translation knowledge described above.

Decision coordination. To ensure a single coherent decision between the SMs, in this case study, the provider specifies and provides non-conflicting evaluation metrics to both Libra and Rainbow. As an initial attempt, the *single-active* control pattern is used to coordinate decisions between Libra and Rainbow, and changes to the system occur through the action component in the shared access infrastructure.

Global configuration incurs a higher computation cost, but results in an optimal configuration. On the other hand, incremental repair incurs a lower computation cost on a shorter timescale, but the resulting decision may not be globally optimal. The natural arrangement is to alternate between the two, performing global configuration only when necessary, and otherwise performing incremental repair to handle the localized problems.

The coordination works as follows: Initially, a user sets up a video conferencing session, and the Libra framework performs the initial conferencing system configuration based on the user request and the provider's recipe. It then informs the Rainbow framework of the architectural model of the system via the model translator, and hands the control to Rainbow. Rainbow begins monitoring the system for problems and making incremental repairs when necessary.

Each time Rainbow detects a problem, it uses its evaluation metrics to determine the best repair. If Rainbow cannot fix the problem, it will pass the control to Libra for total reconfiguration. Libra then composes a new video conferencing session using its own set of evaluation metrics.

Although both frameworks access information about the system, their difference in expertise means different kinds of information are needed. The Libra framework queries system information on demand so that it can make informed, globally optimal decisions. On the contrary, Rainbow needs access to information quickly to make expedient, run-time decisions so it must cache observed or queried states over a window of time, thereby trading off accuracy and overhead (of data-caching) for a shorter latency of adaptation response. Therefore, within the general coordination approach, customization will be necessary depending on the SMs being coordinated.

6 Prototype Implementation

In the previous section, we described at an abstract level how we realized the coordination architecture for the case study to integrate Rainbow and Libra. In this section, we describe the prototype implementation.

Self-management module components. Tailor and the model manager comprise the evaluation and decision components in the Rainbow module. The model manager maintains an architectural model and provides a set of interfaces for updating, changing, and querying the model, and for querying the system. It evaluates the model to detect problems and triggers Tailor to repair them.

Synthesizer is the decision component in the Libra framework. Our prototype implementation contains a specialized synthesizer for the video conferencing service.

Coordination architecture components. The high-level components in the architecture are implemented in Java and provide RMI interfaces for interaction with each other. The communication between components is via XML request and response messages over RMI, where we have defined message schemas for all communication to validate the messages.

The environment measurement mechanism in the system access infrastructure includes two subcomponents—network measurement and monitoring gauges and probes. The network measurement currently provides latency estimates of network connections. This is achieved by using the global network positioning (GNP) [25] approach, which models the Internet as a geometric space. The monitoring gauges and probes use the Siena publish-subscribe infrastructure [4], and currently include probes that can monitor the load and liveness of components in the system.

For resource discovery, we use the network-sensitive service discovery (NSSD) infrastructure [18]. The Libra or Rainbow module can look for a service component by sending a request to the NSSD directory, specifying the desired service type and a set of predicates, which indicates constraints and preferences on the values of certain attributes of the service. Such attributes might include, for example, cost, supported protocols, and even network latency to a particular user.

The action component provides the mechanism for the Libra or Rainbow module to modify the system configuration. It comprises a high-level Java part to provide an RMI interface to the rest of the infrastructure, and a low-level C++ part to interact with the target system and carry out the actual operations.

For the translation infrastructure, the translation repository is realized by a complex hash data structure and an RMI interface for the various translators to store and retrieve the mappings. The data stored in the mappings are Java object representations of the types, elements, operations, errors. As an implementation artifact, we merged the functionalities of the environment measurement and resource discovery translators.

Video conferencing applications. In the video conferencing system, the user applications include NetMeeting and Vic/SDR, which are existing applications, and the handheld conferencing application, which is a slightly-modified version of Vic.

We use the following three system components. The video conferencing gateway (VGW) supports interoperability between H.323-based and SIP-based conferencing applications, i.e., it translates the session negotiations and forwards the video streams during the session [17]. We implemented a handheld proxy (HHP) that is able to join a conferencing session on behalf of a handheld user. Finally, we use a "proxy-based" variant of the end system multicast (ESM) approach described in [6], i.e., the ESM proxies (ESMPs) establish a multicast overlay among themselves, and each end system communicates with a particular ESMP to send and receive multicast data.

Since all these applications are legacy components, we implemented wrappers to allow the system access infrastructure to control these components. Finally, note that most infrastructure components such as the environment measurement and resource discovery mechanisms might be reusable in other efforts of self-management coordination.

7 Evaluation

We use the video conferencing system described in Section 5.1 to test the prototype and demonstrate that a coordination framework designed and implemented according to the recipes of the coordination architecture can consistently and coherently manage a system. To do so, we performed a case study of two adaptation scenarios: one to add a new user and another to respond to component failure. Each required the interaction of the Libra and Rainbow modules to compose and adapt the video conferencing system. The coordinated Libra and Rainbow did achieve the desired adaptations, allowing the new user to join and recovering in the presence of the component failure.

We also used the two scenarios above to quantify the additional overhead introduced by the coordination architecture, i.e., the overhead of the translation infrastructure (but not the system access mechanisms because they would be needed even without coordination).

Scenario 1: *New Vic user*. In the first scenario, a new Vic user requests to join an already running video conferencing session. This request causes the Rainbow model manager to trigger Tailor to perform an adaptation as follows. First, Tailor (through the query translator) queries network measurement for the latency between the new Vic component and each of the three existing ESMPs. After the three queries, Tailor chooses the ESMP that is "closest" to the new Vic and issues a "connect" operation, which goes through the repair translator and is executed by the action component to configure and start the user's conferencing application. Our measurements show that this adaptation takes about 2130 ms, of which the system access mechanisms cost 1600 ms,

the Rainbow module spent 230 ms, and the overhead introduced by translation is 300 ms.

Scenario 2: *Failed ESMP*. In the second scenario, one of the three existing ESMPs in the session fails. This failure is detected by a liveness probe and reported to the model manager, which triggers Tailor to perform an adaptation as follows. First, Tailor (through the model manager and query translator) queries the resource discovery mechanism to find a new ESMP to replace the failed one. Then, Tailor issues the following operations, which are translated by the repair translator and executed by the action component: *shutdown* the failed ESMP, *start* the new ESMP, *connect* the new ESMP to the two remaining ESMPs and the end point served by the failed ESMP (three connect operations), and finally *activate* the new ESMP. The measurement results show that this adaptation takes about 2730 ms, of which 1500 ms are spent on system access mechanisms and 330 ms on Rainbow, and the translation overhead is 900 ms.

From these preliminary results, we believe that the overhead introduced by the translation layer is reasonable, given that in both cases it is well below the cost of actually accessing the system.

8 Discussion

The self-management approach implied in this paper rests on an important assumption that for any target system, the framework has access to some measurement, resource discovery, and effecting mechanisms to observe and change that system. We believe this assumption is reasonable because a growing number of measurement tools and infrastructures are able to provide information about common component or connector properties, such as network bandwidth and latency measurements. Several different resource discovery protocols and infrastructure exist that can discover new services and resources for a system. Likewise, effector technologies are emerging to support dynamic changes to running system components (e.g., Workflakes [15]). For many legacy systems, it is conceivable to use wrappers to add hooks for making system changes.

In order for the coordination approach to work, a few unresolved issues remain. First, to maintain consistency across n SMs, $O(n^2)$ possible paths of translation exist between the SMs and between each SM and the system, potentially requiring $O(n^2)$ translators and complicating integration. However, the number of dimensions, i.e., size of n, is unlikely to be large. Also, the models might not overlap significantly because they address different dimensions of the system. Finally, more sophisticated representation of translation knowledge can help reduce the number of translators to $O(n)$.

Second, there are other self-management dimensions in addition to what we explored in the case study. The coordination of these potentially conflicting dimensions ultimately depends on the policy set forth by the stakeholders. For example, domain experts coordinating the performance and security dimensions of a system might stipulate that security overrides performance in all cases. As long as the coordination policy can be expressed as a set of compatible metrics and enforced by a control pattern, then we believe that our approach would still be applicable.

Third, many different control patterns are possible to coordinate self-management actions. The sophisticated control patterns enable the SMs to engage in intricate interactions. For example, the *democracy* pattern might be used to coordinate five equivalent SMs to increase fault tolerance. Or, the *balance of power* pattern might be used so that a security-oriented SM could veto a potentially insecure change from a performance-oriented SM.

Finally, the work presented in this paper is inherently centralized, where monitoring and action are performed in a centralized fashion within the shared infrastructures. Making this assumption has allowed us to focus on core issues of self-management, namely monitoring, detection, evaluation, and action. At the same time, there may be concerns of scalability and single point of failure. However, the coordination architecture is potentially applicable in a distributed setting. For example, the self-management modules might be distributed in different hosts. One might implement the system access and environment measurement infrastructure using distributed middleware. The translation infrastructure can be distributed by replicating the translators that were shared between SMs, and by replicating the repository to ensure available access of translation knowledge. The coordination and other distributed computing issues are future research problems.

9 Conclusions and Future Work

In this paper we presented a coordination architecture and approach that address the challenges of composing multiple self-management modules in a consistent and coherent manner to manage a system. We demonstrated our approach by integrating two self-management modules, Libra and Rainbow, in a case study, and applying a prototype implementation to an example video conferencing system.

We then showed that the approach works using two adaptation scenarios. Finally, our evaluation based on the prototype shows that the coordination architecture achieves reasonable performance. We believe that our approach is more generally applicable to coordinating multiple self-configuring and self-healing modules, but unresolved issues remain, as we have discussed above.

For future work, we plan to address the difficult challenge of coordinating decisions between different and possibly conflicting SMs. To resolve this challenge, we will need to better characterize the kinds of SMs possible, develop a technique to coordinate the various evaluation met-

rics, and determine a coordination policy to ensure coherent action among the SMs.

We also plan to expand our case study as follows. (1) We will explore the applicability of our approach to the self-optimizing and self-protecting capabilities from IBM's classification. (2) We will add reconfiguration mechanisms to the prototype and conduct a more comprehensive evaluation to include total reconfiguration at system run-time and incorporate greater use of utility evaluation in the coordination between Rainbow and Libra. (3) We will examine performance bottlenecks in the coordination architecture and attempt to optimize our prototype.

Acknowledgments

The research described in this paper was supported 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] Introduction to the MBone. http://www-itg.lbl.gov/mbone/.

[2] Mbone Conferencing Applications. http://www-mice.cs.ucl.ac.uk/multimedia/software/.

[3] *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, Dec. 12–14 2001.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for self-repair. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development, and Maintenance*, pages 45–59, Montral, Qubec, Canada, Aug. 25–30 2002. Kluwer Academic Publishers.

[6] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[7] N. Combs and J. Vagel. Adaptive mirroring of system of systems architectures. In Garlan et al. [11], pages 96–98.

[8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In Garlan et al. [11], pages 21–26.

[9] A. G. Ganak and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[10] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, New York, NY, USA, 2003. Springer-Verlag Inc.

[11] D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the First ACME SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, Charleston, SC, USA, Nov. 18–19 2002. ACM Press.

[12] I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architectures for distributed systems. In Garlan et al. [11], pages 33–38.

[13] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Computer Networks, Special Issue on Pervasive Computing*, 35(4), Mar. 2001.

[14] P. N. Gross, S. Gupta, G. E. Kaiser, G. S. Kc, and J. J. Parekh. An active events model for systems monitoring. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture* [3].

[15] P. N. Gross, S. Gupta, G. E. Kaiser, G. S. Kc, and J. J. Parekh. An active events model for systems monitoring. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture* [3].

[16] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543, IETF, Mar. 1999.

[17] J.-C. Hu and J.-M. Ho. A Conference Gateway Supporting Interoperability Between SIP and H.323 Clients. Master's thesis, Carnegie Mellon University, Mar. 2000.

[18] A.-C. Huang and P. Steenkiste. Network-Sensitive Service Discovery. In *Proc. USITS '03 (to appear)*, Mar. 2003.

[19] A.-C. Huang and P. Steenkiste. Building Self-configuring Services Using Service-specific Knowledge. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (to appear)*, June 2004.

[20] ITU-T Recommendation H.323. Packet-based Multimedia Communications Systems, Nov. 2000.

[21] J. Jann, L. M. Browning, and R. S. Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal*, 42(1):29–37, 2003.

[22] J. C. Knight, D. Heimbigner, A. L. Wolf, A. Carzaniga, J. C. Hill, P. Devanbu, and M. Gertz. The Willow survivability architecture. In *Fourth Information Survivability Workshop*, Vancouver, British Columbia, Oct 2001. Postponed to March 2002.

[23] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.

[24] Microsoft Windows NetMeeting. http://www.microsoft.com/windows/netmeeting/.

[25] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. *INFOCOM '02*, June 2002.

[26] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptative software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

[27] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. *WWW2002 (Web Engineering Track)*, May 2002.

[28] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated Planning for Open Architectures. In *Proc. OPENARCH 2000 – Short Paper Session*, pages 17–20, Mar. 2000.