Chapter 4

TRANSPARENT SHAPING: A METHODOLOGY FOR ADDING ADAPTIVE BEHAVIOR TO EXISTING SOFTWARE SYSTEMS AND APPLICATIONS

S. MASOUD SADJADI
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
sadjadi@cs.fiu.edu

PHILIP K. MCKINLEY and BETTY H.C. CHENG*
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824, USA
{mckinley,chengb}@cse.msu.edu

The need for adaptability in software is growing, driven in part by the emergence of pervasive and autonomic computing. In many cases, it is desirable to enhance existing programs with adaptive behavior, enabling them to execute effectively in dynamic environments. In this chapter, we introduce an innovative software engineering methodology called *transparent shaping* that enables dynamic addition of adaptive behavior to existing software systems and applications. We describe an approach to implementing transparent shaping that combines four key software development techniques: aspect-oriented programming to realize separation of concerns at development time, behavioral reflection to support software reconfiguration at run time, component-based design to facilitate independent development and deployment of adaptive code, and adaptive middleware to encapsulate the adaptive functionality. After presenting the general methodology, we discuss two specific realizations of transparent shaping that we have developed and used to create adaptable systems and applications from existing ones.

1. Introduction

A software application is *adaptable* if it can change its behavior dynamically (at run time) in response to transient changes in its execution

*Corresponding author.

environment or to permanent changes in its requirements. Recent interest in designing adaptable software is driven in part by the emergence of pervasive computing and the demand for autonomic computing.[1] *Pervasive computing* promises anywhere, any time access to data and computing resources with few limitations and disruptions.[2] The need for adaptability in pervasive computing is particularly evident at the "wireless edge" of the Internet, where software in mobile devices must balance conflicting concerns such as quality-of-service (QoS) and energy consumption when responding to variability of conditions (e.g., wireless network loss rate). *Autonomic computing*[3] refers to self-managed, and potentially self-healing, systems that require only high-level human guidance. Autonomic computing is critical to managing the myriad of sensors and other small devices at the wireless edge, but also in managing large-scale computing centers and protecting critical infrastructure (e.g., financial networks, transportation systems, power grids) from hardware component failures, network outages, and security attacks.

Developing and maintaining adaptable software are nontrivial tasks. An adaptable application comprises *functional* code that implements the business logic of the application and supports its imperative behavior, and *adaptive* code that implements the adaptation logic of the application and supports its adaptive behavior. The difficulty in developing and maintaining adaptable applications is largely due to an inherent property of the adaptive code, that is, the adaptive code tends to *crosscut* the functional code. Example crosscutting concerns include QoS, mobility, fault tolerance, recovery, security, self auditing, and energy consumption. Even more challenging than developing new adaptable applications is enhancing *existing* applications, such that they execute effectively in new, dynamic environments not envisioned during their design and development. For example, many non-adaptive applications are being ported to mobile computing environments, where they require dynamic adaptation.

This chapter describes a new software engineering methodology, called *transparent shaping*, that supports the design and development of adaptable programs from existing programs without the need to modify the existing programs' source code directly. We argue that *automatic* generation of an adaptable program from a non-adaptable one is important to maintaining program integrity, not only because it avoids errors introduced by manual changes, but because it provides traceability for the adaptations and enables the program to revert back to its original behavior if necessary. Our approach to implementing transparent shaping combines

four key technologies: *aspect-oriented programming* to enable separation of concerns at development time, *behavioral reflection* to enable software reconfiguration at run time, *component-based design* to enable independent development and deployment of adaptive code, and *adaptive middleware* to help insulate application code from adaptive functionality. To demonstrate the effectiveness of this approach, we describe two realizations of transparent shaping that we have developed and used to create adaptable applications.

The remainder of this chapter is organized as follows. Section 2 discusses the four main components of our approach. Section 3 provides an overview of transparent shaping and describes its relationship to program families.[4] Sections 4 and 5, respectively, describe two realizations of transparent shaping; one is middleware-based and the other is language-based. Section 6 discusses how transparent shaping complements other research in adaptive software. Section 7 presents conclusions and identifies several directions for future research.

2. Basic Elements

Transparent shaping integrates four key technologies: separation of concerns, behavioral reflection, software components, and middleware. In this section, we briefly review each technology and its role in transparent shaping.

Separation of concerns[5] enables the separate development of the functional code from the adaptive code of an application. This separation simplifies development and maintenance, while promoting software reuse. Moreover, since adaptation often involves crosscutting concerns, this separation also facilitates transparent shaping. In our approach, we use aspect-oriented programming (AOP),[6,7] an increasingly common approach to implementing separation of concerns in software. While object-oriented programming introduces abstractions to capture commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, thus complicating the development and maintenance of applications. Conversely, in AOP the code implementing such crosscutting concerns, called *aspects*, is developed separately from other parts of the system. Later, for example during compilation, an *aspect weaver* can be used to weave different aspects of the program together to form a program with new behavior. Predefined locations in the program where aspect code can be woven are called *pointcuts*.

In traditional AOP, after compilation the aspects are tangled (via weaving) with the functional code. To facilitate dynamic reconfiguration, transparent shaping needs a way to enable separation of concerns to persist into run time. This separation can be accomplished using *behavioral reflection*,[8] the second key technology for transparent shaping. Behavioral reflection enables a system to "open up" its implementation details at run time.[9] A reflective system has a self representation that deals with the computational aspects (implementation) of the system, and is *causally connected* to the system. The self-representation of a reflective system is realized by metaobjects residing in the metalevel, which is separated from the actual system represented by objects in the base level. A *metaobject* is an entity that manipulates, creates, describes, or implements other objects, which in turn called *base objects*, and might store some information about these base objects such as their type, interface, class, methods, attributes, variables, functions, and control structures. By incorporating crosscutting concerns associated with the system as part of its self-representation, the resulting code at run time is not tangled and therefore can be reconfigured dynamically. When combined with AOP, behavioral reflection enables dynamic weaving of crosscutting concerns into an application at run time.[10]

The third major technology that supports transparent shaping is component-based design. *Software components* are software units that can be independently developed, deployed, and composed by third parties.[11] Well-defined interface specifications supported in component-based design enable adaptive code to be developed independently from the functional code, and potentially by different parties, using the interface as a contract. Component-based design supports two types of composition. In static composition, a developer can combine several components at compile time to produce an application. In dynamic composition, the developer can add, remove, or reconfigure components within an application at run time. When combined with behavioral reflection, component-based design enables a "plug-and-play" capability for adaptive code to be incorporated with functional code at run time that facilitates development and maintenance of adaptable software.

Finally, in many cases it is desirable to hide the adaptive behavior from the application using middleware. Traditionally, *middleware* is intended to mask the distribution of resources across a network and hide differences among computing platforms and networks.[12] As observed by several researchers,[13] however, middleware is also an ideal place to incorporate

adaptive behavior for many different crosscutting concerns. *Adaptive* middleware enables dynamic reconfiguration of middleware services while an application is running, adjusting the middleware behavior to environmental changes dynamically. Our approach to transparent shaping uses adaptive middleware in two ways. In the first, transparent shaping adds adaptive behavior to a middleware platform already supporting the application. In the second, transparent shaping is used to weave calls to adaptive middleware into an application.

3. General Approach

By generating adaptable programs from existing ones, transparent shaping is intended to support the reuse of those applications in environments whose characteristics were not necessarily anticipated during the original design and development. Therefore, a challenge in transparent shaping is finding a way to produce adaptable programs that share the business logic of the original program and differ only in the new adaptive behavior.

As illustrated in Fig. 1, one way to formulate this problem is using *program families*, a well-established concept in the software engineering community. A program family[4] is a set of programs whose extensive commonalities justify the expensive effort required to study and develop them as a whole, rather than individually. In short, transparent shaping can be viewed as a methodology that produces a family of adaptable programs from an existing non-adaptable program. The adaptable program comprises the original program code that remains fixed during program execution, and adaptive code that can be replaced with other adaptive

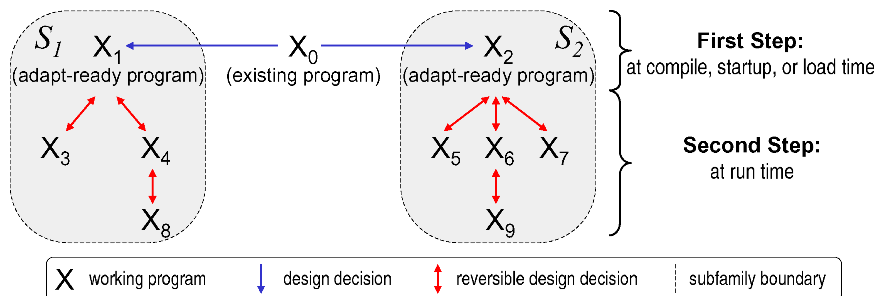

Fig. 1. A transparent shaping design tree illustrating a family of adaptable programs produced from an existing program, which is the root of this tree. Children of the root are adapt-ready programs. Other descendants are adaptable programs.

code dynamically. Replacing one piece of adaptive code with another piece of adaptive code converts an adaptable program into another adaptable program in the corresponding family. This conversion is possible in this programming model, because the adaptive code is not tangled with the functional code. We use the term *composer* to refer to the entity that performs this conversion. The composer might be a human — a software developer or an administrator interacting with a running program through a graphical user interface — or a piece of software — a dynamic aspect weaver, a component loader, a run-time system, or a metaobject.

Transparent shaping produces adaptable programs in two steps. In the first step, an *adapt-ready* program[14] is produced at compile, startup, or load time using static transformation techniques. An adapt-ready program is a program whose behavior is initially equivalent to the original program, but which can be adapted at run time by insertion or removal of adaptive code at certain points in the execution path of the program, called *sensitive joinpoints*. To support such operations, the first step of transparent shaping weaves interceptors, referred to as *hooks*, at the sensitive joinpoints, which may reside inside the program code itself, inside its supporting middleware, or inside the system platform. Example techniques for implementing hooks include aspects (compile time), CORBA portable interceptors[15] (startup time), and byte-code rewriting[16] (load time).

In the second step, executed at run time, the hooks in the adapt-ready program are used by the composer to convert the adapt-ready program into an adaptable program in the corresponding subfamily, as conditions warrant. Adapt-ready programs derived from the same existing program differ in their corresponding sensitive joinpoints and hooks. We note that the available hooks in an adapt-ready program limit its dynamic behavior. In other words, each adapt-ready program can be converted to a limited number of adaptable programs in the corresponding family. The adaptable programs derived from an adapt-ready program form a subfamily (e.g., $S1$ and $S2$ in Fig. 1).

We use Fig. 1 to describe a specific example. Consider an existing distributed program (X_0) originally developed for a wired and secure network. To enable this program to run efficiently in a mobile computing environment, the first step of transparent shaping can be used to produce an adapt-ready version of this program (X_1), which has hooks intercepting all the remote interactions. At run time, if the system detects a low quality wireless connection, the composer can insert adaptive code for tolerating long periods of disconnection into the adapt-ready program

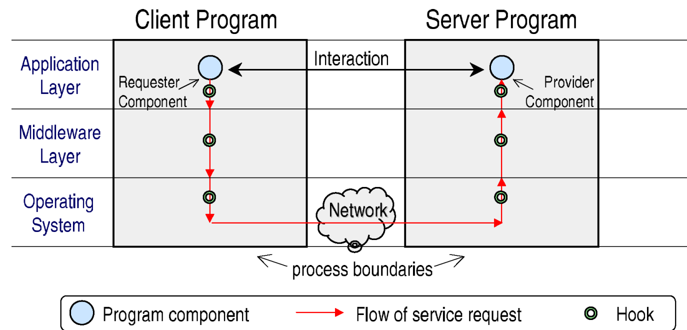

Fig. 2. Alternative places to insert hooks.

(producing X_4 from X_1). Later, if the user enters an insecure wireless network, the composer can insert adaptive code for encryption/decryption of the remote interactions into the program (producing X_8 from X_4). Finally, when the user returns to an area with a secure and reliable wireless connection, the composer can remove the adaptive code for both security and connection-management to avoid unnecessary performance overhead due to the adaptive code (producing X_4 from X_8 and X_1 from X_4, respectively).

We identify three approaches to realize transparent shaping that differ according to the placement of hooks (see Fig. 2): (1) hooks can be incorporated inside an application program itself, (2) inside its supporting middleware, or (3) inside the system platform (operating system and network protocols). A number of projects on cross-layer adaptation use the third approach.[17–19] In this paper, we consider only the first two methods, where the hooks are incorporated either inside the middleware or inside the application. Next, we describe two concrete realizations of each type of transparent shaping. The first, described in Section 4, is a middleware-based approach that uses CORBA portable interceptors[15] as hooks. The second, described in Section 5, uses a combination of aspect weaving and metaobject protocols to introduce dynamic adaptation to the application code directly. Both realizations adhere to the general model described above.

4. Middleware-Based Transparent Shaping

The first realization of transparent shaping we describe is the *Adaptive CORBA Template* (*ACT*),[20,21] which we developed to enable dynamic

adaptation in existing CORBA programs. CORBA was one of the first widely used middleware platforms introduced more than 17 years ago. It is still commonly used in numerous systems.

ACT enhances CORBA to support dynamic reconfiguration of middleware services transparently, not only to the application code, but also to the CORBA code itself. As a realization of transparent shaping, ACT produces an adapt-ready version of an existing CORBA program by introducing a hook to intercept all CORBA remote interactions. Specifically, ACT uses CORBA portable interceptors,[15] which can be incorporated into a CORBA program at startup time using a command-line parameter. Later at run time, these hooks can be used to insert adaptive code into the adapt-ready program, which in turn can adapt the requests, replies, and exceptions passing through the CORBA Object Request Brokers (ORBs). In this manner, ACT enables run-time improvements to the program in response to unanticipated changes in its execution environment, effectively producing other members of the adaptable program family dynamically.

4.1. *ACT Architectural Overview*

Figure 3 shows the flow of a request/reply sequence in a simple CORBA application using ACT. For clarity, CORBA ORB details such as stubs and skeletons are not shown. ACT comprises two main components: a

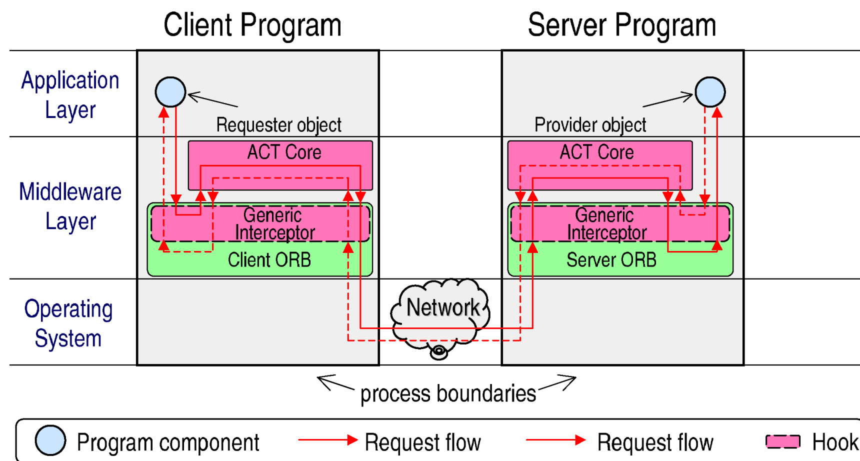

Fig. 3. ACT configuration in the context of a simple CORBA application.

generic interceptor and an ACT core. A *generic interceptor* is a specialized request interceptor that is registered with the ORB of a CORBA application at startup time. The *client* generic interceptor intercepts all outgoing requests and incoming replies (or exceptions) and forwards them to its ACT core. Similarly, the *server* generic interceptor intercepts all the incoming requests and outgoing replies (or exceptions) and forwards them to its ACT core. A CORBA application is called *adapt-ready* if a generic interceptor is registered with all its ORBs at startup time. If, in addition to the generic interceptors, all the ACT core components are also loaded into the application, the application is called *ACT-ready*. Making the application ACT-ready can be done either at startup time or at run time.

4.2. *ACT Core Components*

Figure 4 shows the flow of a request/reply sequence intercepted by the client ACT core. The components of the core include dynamic interceptors, a proxy, a decision maker, and an event mediator. Each component is described in turn.

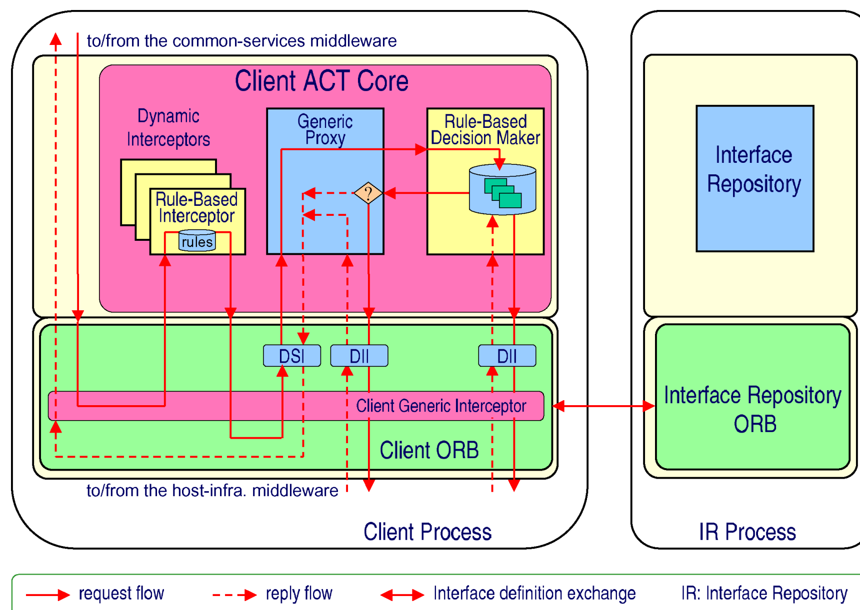

Fig. 4. ACT core components interacting with the rest of the system.

Dynamic Interceptors. According to the CORBA specification,[15] a request interceptor is required to be registered with an ORB at the ORB initialization time. The ACT core enables registration of request interceptors after the ORB initialization time (at run time) by publishing a CORBA interceptor-registration service. Such request interceptors are called *dynamic interceptors*. Dynamic interceptors can be unregistered with the ORB at run time also. In contrast, a request interceptor that is registered with the ORB at startup time is called a *static interceptor* and cannot be unregistered with the ORB during run time. We note that the code developed for a static interceptor and that for a dynamic interceptor can be identical, the difference being the time at which they are registered. In ACT, only generic interceptors are static.

A *rule-based interceptor* is a particular type of dynamic interceptor that uses a set of rules to direct the operations on intercepted requests. The rules can be inserted, removed, and modified at run time. A *rule* consists of two objects: a condition and an action. To determine whether a rule matches a request, a rule-based interceptor consults its condition object. Once a match is found, the interceptor sends the request to the action object of the rule. Since it is part of a CORBA portable interceptor, the action object cannot itself reply to the request or modify the request parameters.[15] The action object can, however, send new requests, record statistics, or raise a ForwardRequest exception, causing the request to be forwarded to another CORBA object such as a proxy.

Proxies. A *proxy* is a surrogate for a CORBA object that provides the same set of methods as the CORBA object. Unlike a request interceptor, a proxy is not prohibited from replying to intercepted requests. A proxy can reply to the intercepted request by sending a new request (possibly with modified arguments) to either the target object or to another object. Alternatively, a proxy can reply to the intercepted requests using local data (e.g., cached replies). However, to enable dynamic weaving of adaptive functionality that is common to multiple CORBA objects, ACT needs to intercept and adapt CORBA requests, replies, and exceptions in a manner independent of the semantics (the application logic) and syntax (the CORBA interfaces defined in the application) of specific applications.

The *generic proxy* is a particular CORBA object that is able to receive *any* CORBA request (hence the label "generic"). To determine how to handle a particular request, the generic proxy accesses the CORBA interface

repository,[15] which provides all the IDL descriptions for CORBA requests. The repository executes as a separate process and is usually accessed through the ORB. Most CORBA ORBs provide a configuration file or support a command-line argument that enables the user to introduce the interface repository to the application ORB. Providing IDL information to the generic proxy in this manner implies no need to modify or recompile the application source code. The interface repository, however, requires access to the CORBA IDL files used in the application.

In default operation mode, the generic proxy intercepts CORBA requests, acquires the request specifications from a CORBA interface repository, creates similar CORBA requests and sends them to the original targets, and forwards replies from those targets back to the original clients. A generic proxy also publishes a CORBA service that can be used to register a *decision maker*.

Decision Makers. A *decision maker* assists proxies in replying to intercepted requests as depicted in Fig. 4. A decision maker receives requests from a proxy and, similar to a rule-based interceptor, uses a set of rules to direct the operation on the intercepted requests. However, unlike a rule-based interceptor, a decision maker is not prohibited from replying to the requests.

4.3. *ACT Operation*

In addition to showing the ACT core components, Fig. 4 also illustrates the sequence of a request/reply inside the ACT core, which contains a rule-based interceptor, a generic proxy, and a rule-based decision maker. First, a request from the client application is intercepted by the rule-based interceptor, which checks its rules for possible matches. A default rule, initially inserted in its knowledge base, directs the rule-based interceptor to raise a ForwardRequest exception, which results in its forwarding the request to the generic proxy. When the generic proxy receives the request, it acquires the request interface definition via the application ORB, which in turn retrieves the information from the interface repository. The proxy creates a new request and forwards it to the rule-based decision maker. The rule-based decision maker checks its knowledge base for possible matches to the request. Depending on the implementation of the rules, the decision maker may return either a modified request to the generic proxy or a reply to the request. If the decision maker returns the request (or a modified

request), then the generic proxy will continue its operation by invoking the request. If the reply to the request is returned by the decision maker, then the proxy replies to the original request using the reply from the decision maker. The generic proxy uses the CORBA dynamic skeleton interface (DSI)[15] to receive any type of request. The generic proxy and the rule-based decision maker use the CORBA dynamic invocation interface (DII)[15] to create and invoke a new request dynamically.

4.4. *ACT/J Implementation*

We have developed an instance of ACT in Java, called *ACT/J*, to evaluate ACT in practice. ACT/J was tested over ORBacus,[22] a CORBA-compliant ORB distributed by IONA Technologies. ORBacus,[22] like JacORB,[23] TAO,[24] and many other CORBA ORBs, supports *CORBA portable interceptors*,[15] which is the only requirement for using ACT.

To make a CORBA application ACT-ready at the application startup time, we need to resolve the following bootstrapping issues. First, we need to register a generic interceptor with the application ORB. Like many other ORBs, ORBacus uses a configuration file that enables an administrator to register a CORBA portable interceptor with the application ORB. JacORB and TAO use a similar approach. Second, since the components in the ACT core are also CORBA objects, they require an ORB to support their operation (registration of services, and so on). Therefore, we need either to obtain a reference to the application ORB for this purpose, or to create a new ORB. ORBacus does provide such a reference, although the CORBA specification does not support this feature. To implement ACT/J over an ORB that does not provide such a reference, we simply create a new ORB, although its use introduces additional overhead.

To test the operation of ACT/J, we developed two administrative consoles: the Interceptor Registration Console and the Rule Management Console. Please note that in this study the composer is assumed to be a human, who performs dynamic adaptation using the administrative consoles. The *Interceptor Registration Console* enables a user to manually register a dynamic interceptor. This console first obtains a generic interceptor name from the user and checks if the generic interceptor is registered with the CORBA naming service. Next, the user can register a dynamic interceptor with the generic interceptor. The *Rule Management Console* allows a user to manually insert rules into rule-based interceptors.

4.5. *ACT/J Case Study*

To evaluate the effectiveness of ACT/J to support self-management in existing CORBA applications, without modifying the application code, we conducted a case study in which self-optimization is enabled in an existing application. Additional experiments involving IP handoff, are described in an accompanying technical report.[25] We begin with a brief overview of the application and the experimental environment, followed by the description of the experiment. The experiment shows how ACT/J could be used to support autonomic computing in either a generic or application-specific manner.

For the application, we adopted an existing distributed image retrieval application developed by BBN Technologies.[26] The application has two parts, a client that requests and displays images, and a server that stores the images and replies to requests for them. In this study, we treat the application as though it were used for surveillance, with a mobile user executing the client code on a laptop and monitoring a physical facility through continuous still images from multiple camera sources. For the experiment described later in this section, we executed the server on a desktop computer connected to a 100 Mbps wired network and the client on a laptop computer connected to a three-cell 802.11b wireless network. Both the desktop and laptop systems are running the Linux operating system.

Figure 5 shows the physical configuration of the three access points used in the experiment. (The wireless cells are drawn as circles for simplicity — the actual cell shapes are irregular, due to the physical construction of the building and orientation of antennas.) AP-1 and AP-3 provide 11 Mbps connections, whereas AP-2 provides only 2 Mbps. The desktop running the server application is close to AP-1. AP-1 and AP-2 are managed by our Computer Science and Engineering Department, whereas AP-3 is managed by the College of Engineering. This difference implies that the IP address assigned to the client laptop needs to change as the user moves from a CSE wireless cell to a College cell. The server provides four different versions of each image, varying in size and quality. Typical comparative file sizes are 90KB, 25KB, 14KB, and 4KB.

To investigate how ACT/J can support self-management, we developed an application-specific rule that maintains the frame rate of the application by controlling the image size or inserting inter-frame delays dynamically. The original image retrieval application operates in a default mode, which retrieves and plays images as fast as possible. ACT/J enables a developer

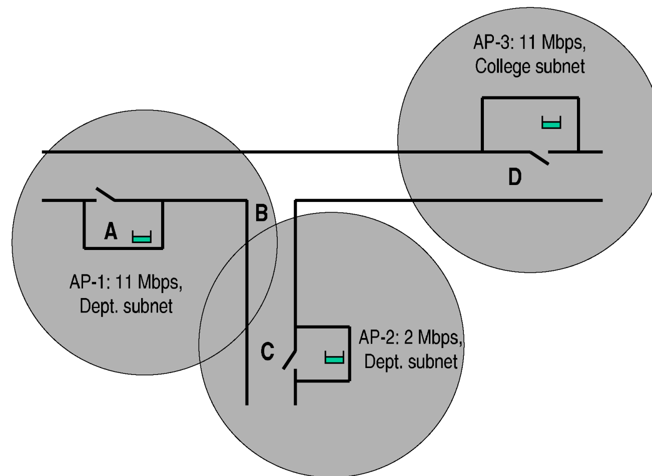

Fig. 5. The configuration of the access points used in the experiment.

to weave the rule into the application at run time, thereby providing new functionality (frame rate control) transparently with respect to the application. The self-optimization rule maintains the frame rate of the application in the presence of dynamic changes to the wireless network loss rate, the network (wired/wireless) traffic, and CPU availability.

We developed a user interface, called the Automatic Adaptation Console, which displays the application status and also enables the user to enter quality-of-service preferences (see Fig. 6). The rule uses several parameters to decide on when and how to adapt the application in order to maintain the frame rate. These parameters have default values as shown in the figure, but can be modified at run time by the user. The `Average Frame Rate Period` indicates the period during which the average frame rate should be calculated to be considered for adaptation. The `Stabilizing Period` specifies the amount of time that the rule should wait until the last adaptation stabilizes; also if a sudden change occurs in the environment such as hand-off from one wireless cell to another one, then the system should wait for this period before it decides on the stability of the system. The rule detects a stable situation using the `Acceptable Rate Deviation`; when the frame rate deviation goes below this value, the system is considered stable. Similarly, the rule detects an unstable situation, if the instantaneous frame rate deviation goes beyond the `Unacceptable Rate Deviation` value. The rule also maintains a history of the round-trip delay

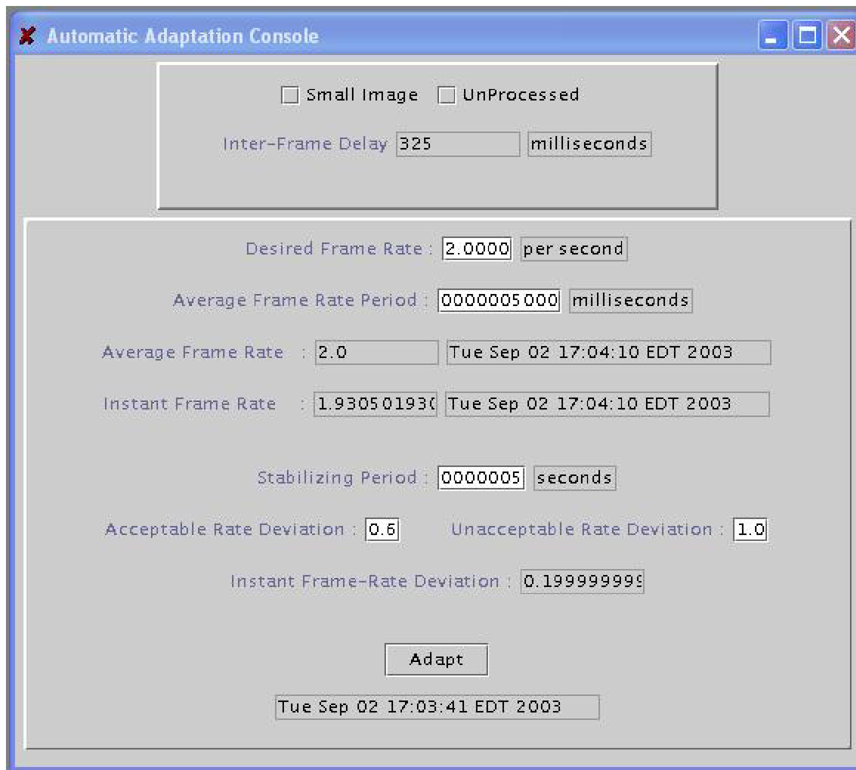

Fig. 6. Automatic Adaptation Console.

associated with each request in each wireless cell. Using this history and the above parameters, the rule can decide to maintain the frame rate either by increasing/decreasing the inter-frame delay or by changing the request to ask for a different version of the image with smaller/larger size. The default behavior of the rule is to display images that are as large as possible, given the constraints of the environment.

Figure 7 shows a trace demonstrating automatic adaptation of the application in the following scenario. In this experiment, the user has selected a desired frame rate of 2 frames per second, as shown in Fig. 6. For the first 60 seconds of the experiment, the user stays close to the location A (Fig. 5). The rule detects that the desired frame rate is lower than the maximum possible frame rate, based on observed round-trip times. Hence, it inserts an inter-frame delay of approximately 200 milliseconds to maintain the frame rate at about 2 frames per second. At point 120 seconds, the user

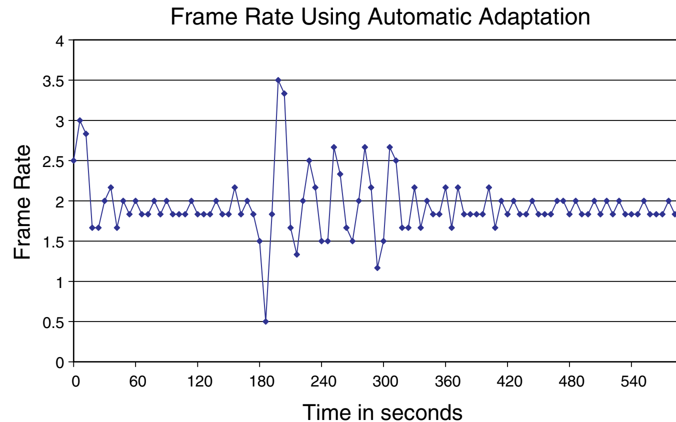

Fig. 7. Maintaining the application frame rate using automatic adaptation.

starts walking from location A to location B for 60 seconds. The automatic adaptation rule maintains the frame rate by decreasing the inter-frame delay during this period. At point 180 seconds, the user begins walking from location B to location C and back again, returning to location B at 360 seconds. During this period, because the AP-2 access point provides 2 Mbps, the automatic adaptation rule detects that the current frame rate is lower than that desired. It first removes the inter-frame delay, but the frame rate does not reach to 2 frames per second. Therefore, it reduces the quality of the image by asking for a smaller image size. Now the frame increases beyond that desired, so the automatic adaptation rule inserts an inter-frame delay of 400 milliseconds to maintain the frame rate at 2 frames per second. Although there is some oscillation, the rate stabilizes by time 360 seconds. At this point, the user continues walking from location B to location A, prompting the rule to reverse the actions. First the inter-frame delay is increased to maintain the frame rate, followed by an increase in image size. In this manner, the rule brings the application back to its original behavior. Again, because the current frame rate is higher than expected, an inter-frame delay of about 200 milliseconds is inserted to maintain the frame rate at 2 frames per second.

This result is promising and demonstrates that it is possible to add self-management behavior to an application transparently to the application code. Moreover, the use of a generic proxy enables self-optimization functionality, both application-independent and application-specific, to be added to the application, even at run time.

As a middleware-based realization of transparent shaping, ACT can be used to produce families of adaptable programs from existing CORBA programs, without the need to modify or recompile their source code. Using the generic interceptor as a hook inside middleware at startup time, ACT enables independent development and deployment of adaptive code from the application code at run time. In ACT, adaptive code are realized as software components (rules and proxies) that can be deployed inside the ACT core dynamically. By allowing dynamic insertion and removal of such adaptive code, ACT enables dynamic conversion of an adapt-ready CORBA program to different adaptable programs in its corresponding program subfamily.

5. Language-Based Transparent Shaping

Although transparent shaping can be realized by incorporating hooks inside middleware, as in ACT, many programs do not use middleware explicitly. In this section, we introduce TRAP (Transparent Reflective Aspect Programming),[27] a language-based realization of transparent shaping that supports dynamic adaptation in existing programs developed in class-based, object-oriented programming languages. TRAP uses generative techniques to create an adapt-ready application, without requiring any direct modifications to the existing programs.

With TRAP, the developer selects at compile time a subset of classes in the existing program that are to be reflective at run time. We say a class is *reflective* at run time if its behavior (e.g., the implementation of its methods) can be inspected and modified dynamically. Since many object-oriented languages, such as Java and C++, do not support such functionality inherently, TRAP uses generative techniques to produce an adapt-ready program with hooks that provide the reflective facilities for the selected classes. As the adapt-ready program executes, new behavior can be introduced to the program by insertion and removal of adaptive code via interfaces to the reflective classes.

5.1. *TRAP/J Architectural Overview*

We developed TRAP/J, a prototype instantiation of TRAP for Java programs.[27] The operation of the first step, converting an existing Java program into an adapt-ready program, is depicted in Fig. 8. We assume that the `.java` source files of the original application are not available. The

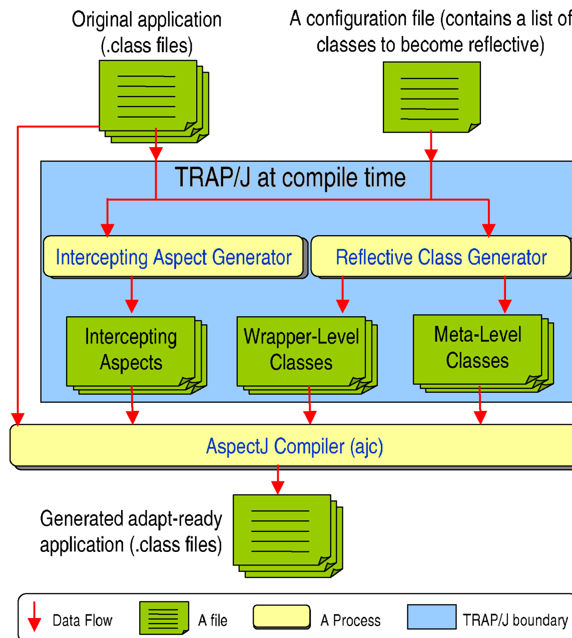

Fig. 8. TRAP/J operation at compile time.

compiled class files (`.class` files) of the application and a configuration file containing a list of class names (the ones selected to be reflective) are input to an Aspect Generator and a Reflective Class Generator. For each class name in the list, these generators produce one aspect, one wrapper-level class, and one metalevel class. Next, the generated aspects and reflective classes, along with the original application compiled class files, are passed to the AspectJ compiler (`ajc`),[28] which weaves the generated and original application code together to produce an adapt-ready application. The second step occurs at run time, when new behavior can be introduced to the adapt-ready application using the wrapper- and meta-level classes (also referred to as the adaptation infrastructure). Specifically, the interface of the metalevel class includes services that enable methods of the wrapper-level class to be overridden at run time with new implementations, called *delegates*.

Figure 9 illustrates the interaction among the Java Virtual Machine (JVM) and the administrative consoles (GUI). First, the adapt-ready application is loaded by the JVM. At the time each metaobject is instantiated, it registers itself with the Java rmiregistry using a unique

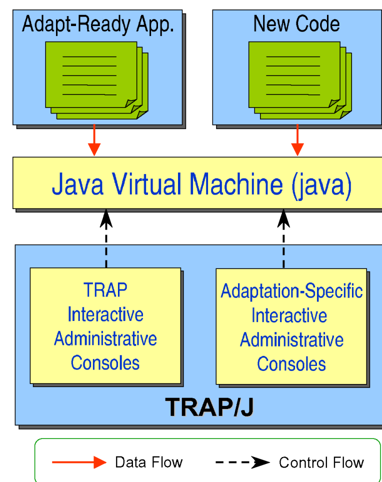

Fig. 9. TRAP/J run-time support.

ID. Next, if an adaptation is required, the composer dynamically adds new code to the adapt-ready application at run time, using Java RMI to interact with the metaobjects. As part of the behavioral reflection provided in the adaptation infrastructure, a metaobject protocol (MOP) is supported in TRAP/J that allows interception and reification of method invocations targeted to objects of the classes selected at compile time to be adaptable.

5.2. *TRAP/J Run-Time Model*

To illustrate the operation of TRAP/J, let us consider a simple application comprising two classes, Service and Client, and three objects, (client, s1, and s2). Figure 10 depicts a simple run-time class graph for this application that is compliant with the run-time architecture of most class-based object-oriented languages. The class library contains Service and Client classes, and the heap contains client, s1, and s2 objects. The "instantiates" relationship among objects and their classes are shown using dashed arrows, and the "uses" relationships among objects are depicted with solid arrows.

Figure 11 illustrates a layered run-time class graph model for this application. Please note that the base-level layer depicted in Fig. 11 is equivalent to the class graph illustrated in Fig. 10. For simplicity, only the "uses" relationships are represented in Fig. 11. The wrapper level contains the generated wrapper classes for the selected subset of base-level classes

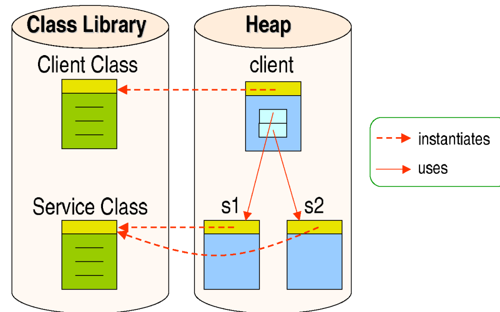

Fig. 10. A simplified run-time class graph.

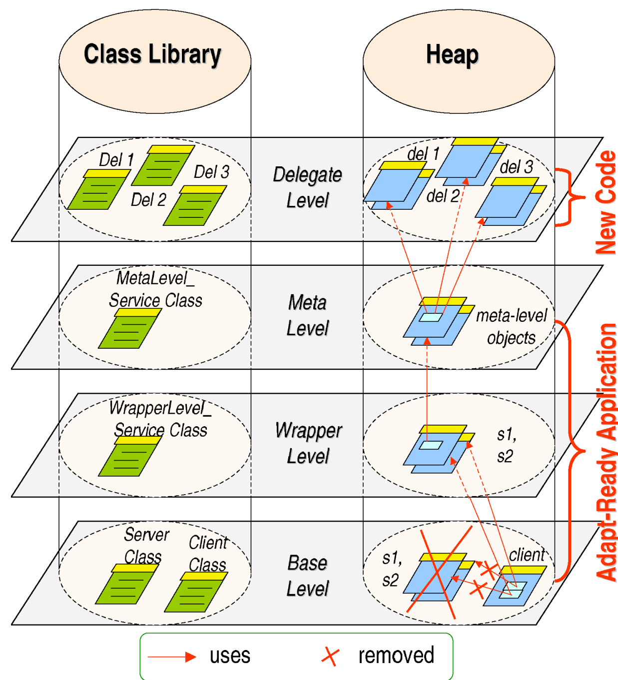

Fig. 11. TRAP layered run-time model.

and their corresponding instances. The base-level client objects use these wrapper-level instances instead of base-level service objects. As shown, `s1` and `s2` no longer refer to objects of the type `Service`, but instead refer to objects of type `ServiceWrapper` class. The metalevel contains the generated metalevel classes corresponding to each selected base-level class and their

corresponding instances. Each wrapper class has exactly one associated metalevel class, and associated with each wrapper object can be at most one metaobject. In this way, the behavior of each object in response to each message is dynamically programmable, using the generic method execution MOP provided in TRAP/J.

Finally, the delegate level contains adaptive code that can dynamically override base-level methods that are wrapped by the wrapper classes. Adaptive code is introduced into TRAP/J using *delegate* classes. A delegate class can contain implementation for an arbitrary collection of base-level methods of the wrapped classes, enabling the localization of a crosscutting concern in a delegate class. A composer can program metaobjects dynamically to redirect messages destined originally to base-level methods to their corresponding implementations in delegate classes. Each metaobject can use one or more delegate instances, enabling different crosscutting concerns to be handled by different delegate instances. Moreover, delegates can be shared among different metaobjects, effectively providing a means to support dynamic aspects.

For example, let us assume that we want to adapt the behavior of a socket object (instantiated from a Java socket class such as the Java.net.MulticastSocket class) in an existing Java program at run time. First, at compile time, we use TRAP/J generators to generate the wrapper and metaobject classes associated with the socket class. Next, at run time, a composer can program the metaobject associated with the socket object to support dynamic reconfiguration. Programming the metaobject can be done by introducing a delegate class to the metaobject at run time. The metaobject then loads the delegate class, instantiates an object of the delegate class, intercepts all subsequent messages originally targeted to the socket object, and forwards the intercepted messages to the delegate object. Let us assume that the delegate object provides a new implementation for the `send()` method of the socket class. In this case, all subsequent messages to the `send()` method are handled by the delegate object and the other messages are handled by the original socket object. Alternatively, the delegate object could modify the intercepted messages and then forward them back to the socket object, resulting in a new behavior. TRAP/J allows the composer to remove delegates at run time, bringing the object behavior back to its original implementation. Thus, TRAP/J is a non-invasive[29] approach to dynamic adaptation.

In an earlier study,[27] we developed a delegate that effectively allows selected Java sockets in an existing program to be replaced with adaptable

communication middleware components called MetaSockets. A *MetaSocket* is created from existing Java socket classes, but its structure and behavior can be adapted at run time in response to external stimuli such as dynamic wireless channel conditions. Specifically, data sent or received on the socket is passed through a pipeline of filters. A MetaSocket itself can be reconfigured dynamically in its filter pipeline. The filter pipeline can be reconfigured dynamically, that is, filters can be inserted and removed, in response to changes in changing conditions. Moreover, the filter components can be developed by third parties and can be independent of the functional code of an application. Using TRAP/J and MetaSockets, we demonstrated how to transform existing network applications into adaptive applications that can better tolerate dynamic conditions on wireless networks.[27]

5.3. *TRAP/J Case Study*

To demonstrate how TRAP/J can be used to produce adaptable programs from an existing program without the need to modify the existing program source code directly, we use the Audio Streaming Application, called ASA, that is designed to stream interactive audio from a microphone at one network node to multiple receiving nodes. The original application was developed for wired networks. Our goal is to adapt this application to wireless environments, where the packet loss rate is dynamic and location dependent.

In this case study, we configured the experiments in an *ad hoc* wireless network as illustrated in Fig. 12. A laptop workstation transmits an audio stream to multiple wireless iPAQs over an 802.11b (11 Mbps) *ad hoc* wireless local area network (WLAN). Please note that unlike in wired networks, in wireless networks factors such as signal strength, interference,

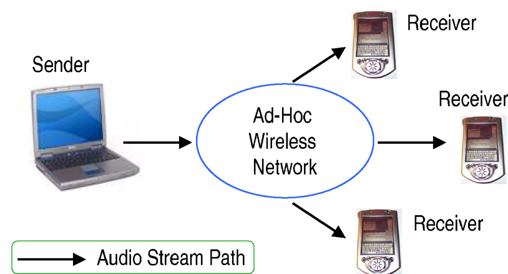

Fig. 12. Audio streaming in a wireless LAN.

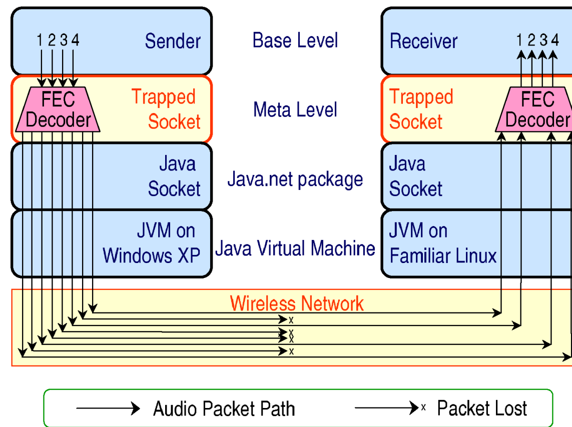

Fig. 13. Adaptation strategy.

and antenna alignment produce dynamic and location-dependent packet losses. In current WLANs, these problems affect multicast connections more than unicast connections, since the 802.11b MAC layer does not provide link-level acknowledgements for multicast frames.

Figure 13 illustrates the strategy we used to enable ASA to adapt to variable channel conditions in wireless networks. However, we used TRAP/J to modify ASA *transparently* such that it uses MetaSockets instead of Java multicast sockets. The particular MetaSocket adaptation used here is the dynamic insertion and removal of *forward-error correction* (FEC) filters.[30] Specifically, an FEC encoder filter can be inserted and removed dynamically at the sending MetaSocket, in synchronization with an FEC decoder being inserted and removed at each receiving MetaSocket. Use of FEC under high packet loss conditions reduces the packet loss rate as observed by the application. Under low packet loss conditions, however, FEC should be removed so as not to waste bandwidth on redundant data.

Making ASA Adapt-Ready. Figure 14 shows excerpted code for the original Sender class. The main method creates a new instance of the Sender class and calls its run method. The run method first creates an instance of AudioRecorder and MulticastSocket and assigns them to the instance variables, ar and ms, respectively. The multicast socket (ms) is used to send the audio datagram packets to the receiver applications. Next, the run method executes an infinite loop that, for each iteration, reads live audio data and transmits the data via the multicast socket.

```
1   public class Sender
2   {
3     AudioRecorder ar;
4     MulticastSocket ms;
5     public void run()
6     { ...
7       ar = new AudioRecorder(...);
8       ms = new MulticastSocket();
9       byte[] buf = new byte[500];
10      DatagramPacket packetToSend =
11        new DatagramPacket(buf, buf.length,
12        target_address, target_port);
13      while (!EndOfStream)
14      {
15        ar.read(buf, 0, 500);
16        ms.send(packetToSend);
17      } // end while ...
18    }
19  } // end Sender
```

Fig. 14. Excerpted code for the Sender class.

Compile-Time Actions. The Sender.java file and a file containing only the java.net.MulticastSocket class name are input to the TRAP/J aspect and reflective generators. The TRAP/J class generators produce one aspect file, named Absorbing_MulticastSocket.aj (for base-level), and two reflective classes, named WrapperLevel_MulticastSocket.java (wrapper level) and MetaLevel_MulticastSocket.java (metalevel). Next, the generated files and the original application code are compiled using the AspectJ compiler (ajc) to produce the adapt-ready program. We note that if ajc could accept .class files instead of .java files, then we would not even need the original source code in order to make the application adapt-ready.

Generated Aspect. The aspect generated by TRAP/J defines an initialization `pointcut` and the corresponding `around` advice for each `public` constructor of the MulticastSocket class. An `around` advice causes an instance of the generated wrapper class, instead of an instance of MulticastSocket, to serve the sender. Figure 15 shows excerpted code for the generated Absorbing_MulticastSocket aspect. This figure shows the "initialization" `pointcut` (lines 3–4) and its corresponding `advice` (lines 6–11) for the MulticastSocket constructor used in the Sender class. Referring back to the layered class graph in Fig. 11, the sender (client) uses an instance

```
1   public aspect Absorbing_MulticastSocket
2   {
3     pointcut MulticastSocket() :
4       call(java.net.MulticastSocket.new()) && ...;
5
6     java.net.MulticastSocket around()
7       throws java.net.SocketException
8       : MulticastSocket()
9     {
10      return new WrapperLevel_MulticastSocket();
11    }
12
13    pointcut MulticastSocket_int(int p0) :
14      call(java.net.MulticastSocket.new(int))
15        && args(p0) && ...;
16
17    // Pointcuts and advices around the   nal public methods
18    pointcut getClass(WrapperLevel_MulticastSocket
19      targetObj) :
20      ...;
21  }
```

Fig. 15. Excerpted generated aspect code.

of the wrapper class instead of the base class. In addition to handling `public` constructors, TRAP/J also defines a `pointcut` and an `around` advice to intercept all `public final` and `public static` methods.

Generated Wrapper-Level Class. Figure 16 shows excerpted code for the WrapperLevel_MulticastSocket class, the generated wrapper class for the MulticastSocket. This wrapper class extends the MulticastSocket class. All the `public` constructors are overridden by passing the parameters to the super class (base-level class) (lines 4–6). Also, all the `public` instance methods are overridden (lines 8–29).

To better explain how the generated code works, we step through the details of how the send method is overridden, as shown in Fig. 16. The generated send method first checks whether the metaObject variable, referring to the metaobject corresponding to this wrapper-level object, is null (lines 11–12). If so, then the base-level (super) method is called, as if the base-level method had been invoked directly by another object, such as an instance of sender. Otherwise, a message containing the context information is dynamically created using Java reflection and passed to the metaobject (metaObject) (lines 14–28). It might be the case that a metaobject may need

```
1   public class WrapperLevel_MulticastSocket extends
2     MulticastSocket implements WrapperLevel_Interface {
3
4     // Overriding the base-level constructors.
5     public WrapperLevel_MulticastSocket()
6       throws SocketException { super(); }
7
8     // Overriding the base-level methods.
9     public void send(java.net.DatagramPacket p0)
10      throws IOException {
11      if(metaObject == null)
12      { super.send(p0); return; }
13      . . .
14      Class[] paramType = new Class[1];
15      paramType[0] = java.net.DatagramPacket.class;
16      Method method = WrapperLevel_MulticastSocket.
17        class.getDeclaredMethod("send", paramType);
18
19      Object[] tempArgs = new Object[1];
20      tempArgs[0] = p0;
21      ChangeableBoolean isReplyReady =
22        new ChangeableBoolean(false);
23
24      try {
25        metaObject.invokeMetaMethod
26          (method, tempArgs, . . . );
27      } catch (java.io.IOException e) { throw e; }
28      catch (MetaMethodIsNotAvailable e) {}
29  }
```

Fig. 16. Excerpted generated wrapper code.

to call one or more of the base-level methods. To support such cases, which we suspect might be very common, the wrapper-level class provides access to the base-level methods through the special wrapper-level methods whose names match the base-level method names, but with an "Orig_" prefix.

Generated Metalevel Class. Figure 17 shows excerpted code for MetaLevel_MulticastSocket, the generated metalevel class for MulticastSocket. This class keeps an instance variable, delegates, which is of type Vector and refers to all the delegate objects associated with a metaobject that implements one or more of the base-level methods. To support dynamic adaptation of the static methods, a metalevel class provides the staticDelegates instance variable and its corresponding insertion and removal methods (not shown). *Delegate* classes introduce new code to applications

```
1   public class MetaLevel_MulticastSocket
2     extends UnicastRemoteObject
3     implements MetaLevel_Interface,DelegateManagement{
4
5     private Vector delegates = new Vector();
6     public synchronized void insertDelegate
7       (int i, String delegateClassName)
8       throws RemoteException { ... }
9     public synchronized void removeDelegate(int i)
10      throws RemoteException { ... }
11
12    public synchronized Object invokeMetaMethod
13      (Method method, Object[] args,
14      ChangeableBoolean isReplyReady) throws Throwable{
15      // Finding a delegate that implements this method
16      ...
17      if(!delegateFound) // No meta-level method available
18        throw new MetaMethodIsNotAvailable();
19      else
20        return newMethod.invoke(delegates.get(i-1),
21          tempArgs);
22  }
```

Fig. 17. Excerpted generated metaobject code.

at run time by overriding a collection of base-level methods selected from one or more of the *adaptable* base-level classes. An adaptable base-level class has corresponding wrapper- and metalevel classes, generated by TRAP/J at compile time. Metaobjects can be programmed dynamically by inserting or removing delegate objects at run time. To enable a user to change the behavior of a metaobject dynamically, the metalevel class implements the DelegateManagement interface, which in turn extends the Java RMI Remote interface (lines 5–10). A composer can remotely "program" a metaobject through Java RMI. The insertDelegate and removeDelegate methods are developed for this purpose.

The metaobject protocol developed for metalevel classes defines only one method, invokeMetaMethod, which first checks if any delegate is associated with this metaobject (lines 12–22). If not, then a MetaMethodIsNotAvailable exception is thrown, which eventually causes the wrapper method to call the base-level method as described before. Alternatively, if one or more delegates is available, then the first delegate that overrides the method is selected, a new method on the delegate is created using Java reflection, and the method is invoked.

Adapting to Loss Rate. To evaluate the TRAP/J-enhanced audio application, we conducted two sets of experiments similar to those in the previous section. The configuration used in these sets of experiments is illustrated in Fig. 12.

In the first sets of experiments, a user holding a receiving iPAQ handheld computer is walking within the wireless cell, receiving and playing a live audio stream. Figure 18 shows a sample of the results. For the first 120 seconds, the program has no FEC capability. At 120 seconds, the user walks away from the sender and enters an area with loss rate around 30%. The adaptable application detects the high loss rate and inserts a (4, 2) FEC filter, which greatly reduces the packet loss rate as observed by the application, and improves the quality of the audio as heard by the user. At 240 seconds, the user approaches the sender, where the network loss rate is again low. The adaptable application detects the improved transmission and removes the FEC filters, avoiding the waste of bandwidth with redundant packets. Again at 360 seconds, the user walks away from the sender, resulting in the insertion of FEC filters. This experiment demonstrates the utility of TRAP/J to transparently and automatically enhance an existing application with new adaptive behavior.

Balancing QoS and Energy Consumption. In the second set of experiments, we used two MetaSocket filters, SendNetLossDetector and RecvNetLossDetector, which cooperate to monitor the raw loss rate of the

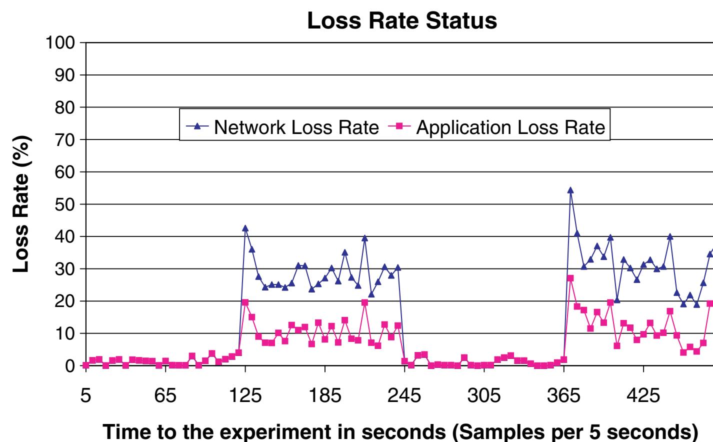

Fig. 18. The effect of using FEC filters to adapt ASA to high loss rates on a wireless network.

wireless channel. Similarly, the SendAppLossDetector and RecvAppLossDetector filters are used to monitor the packet loss rate as observed by the application, which may be lower than the raw packet loss rate due to the use of FEC. At present, a simple state machine is used by a decision maker (DM) component to govern changes in filter configuration. For example, if the loss rate observed by the application rises above a specified threshold, then the DM decides to insert an FEC filter in the pipeline. In case an FEC filter is already present in the pipeline, DM decides to modify the (n, k) parameters of the FEC filter to increase improve QoS. On the other hand, if the raw packet loss rate on the channel drops below a lower threshold, then the level of redundancy is decreased by modifying the parameters of the FEC filter, or in case the FEC filter is not required anymore, DM removes the FEC filter entirely.

Figure 19 shows a trace of an experiment using the ASA described earlier, running in *ad hoc* mode. A stationary user speaks into a laptop microphone, while another user listens on an iPAQ as he changes his location in the wireless cell over a period of time. In this particular test, the iPAQ user remains in a low packet loss area for approximately 30 minutes, moves to a high packet loss area for another 40 minutes, moves back to the low packet loss location for another 30 minutes, and then re-enters the high packet loss location. The user remains there until the iPAQ's external battery drains and the network is disconnected.

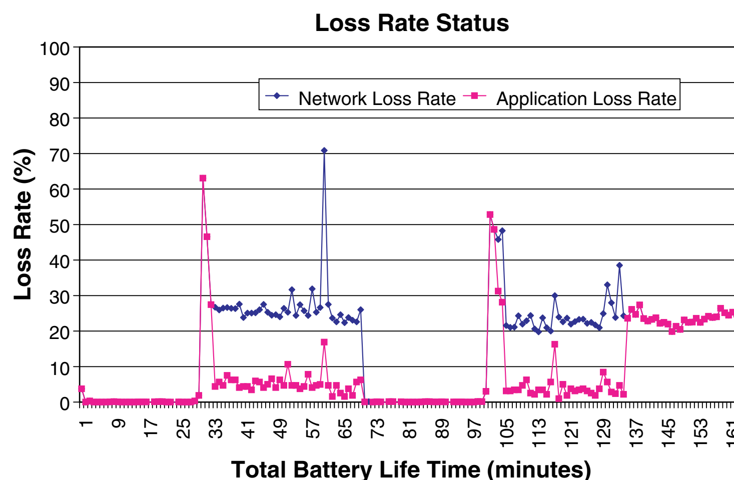

Fig. 19. MetaSocket packet loss behavior with dynamic FEC filter insertion and removal.

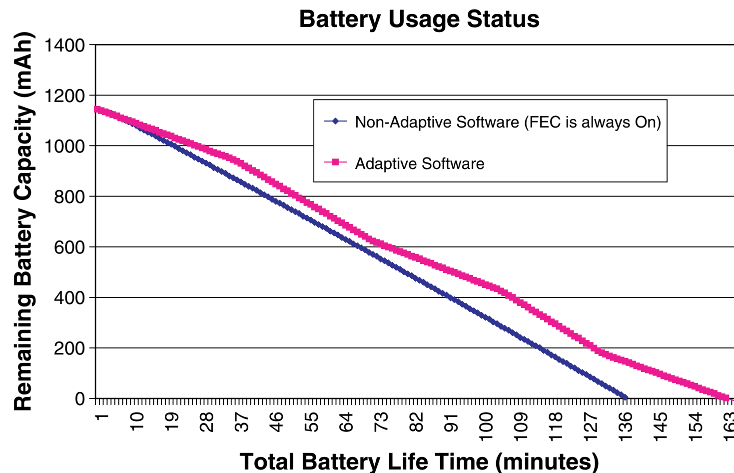

Fig. 20. Trace of energy consumption during experiment using a software measurement technique.

In this experiment, the upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 20%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 5%. As shown in Fig. 19, the FEC $(4, 2)$ code is effective in reducing the packet loss rate as observed by the application. Figure 20 plots the remaining battery capacity as measured during the above experiment and that for a non-adaptive trace. The adaptive version extends the battery lifetime by approximately 27 minutes.

In summary, TRAP enables production of adaptable program families from existing programs developed in class-based, object-oriented programming languages. Using the wrapper- and metalevel classes as hooks instrumented inside the application code at compile time, TRAP enables separate development and deployment of adaptive code in existing programs at run time. In TRAP, pieces of adaptive code are realized as delegates that can be inserted into and removed from an adapt-ready program dynamically, thereby converting the adapt-ready program to adaptable programs in its corresponding program subfamily.

6. Discussion

Figure 21 summarizes the current status of transparent shaping realizations. We have implemented and tested ACT/J and TRAP/J, as described above.

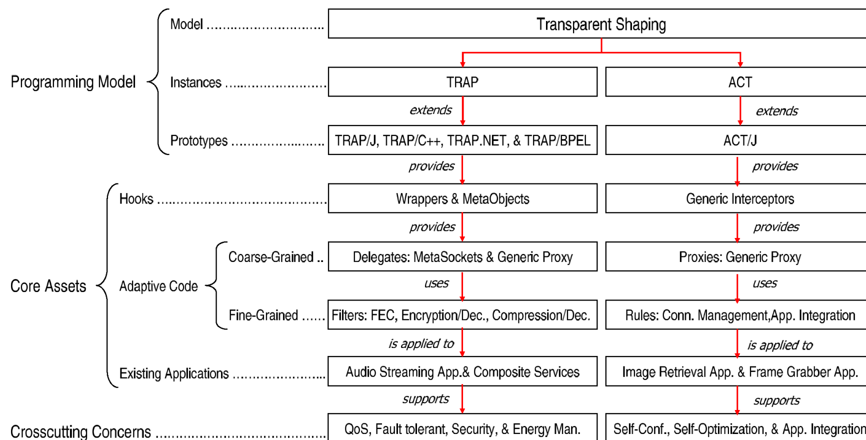

Fig. 21. Transparent shaping status.

We have also developed several core assets for supporting transparent shaping, including examples of hooks, adaptive code, and existing applications. The hooks in TRAP/J are pairs of wrappers and metaclasses, which are generated by TRAP/J generators automatically. In ACT/J, there is only one hook, the generic portable interceptor, which can be reused in any CORBA program. Adaptive code in TRAP/J is realized by delegates. A reusable delegate using MetaSockets and filters is provided. A generic proxy was developed for ACT/J that can be used in any existing CORBA application. The generic proxy can receive any CORBA request and can adapt it using adaptive code realized by rules.

We are currently addressing several other aspects of transparent shaping. To support existing programs developed in C++, .NET, and BPEL, members of our group have already implemented TRAP/C++[31] using compile-time metaobject protocols supported by Open C++,[32] TRAP.NET[33] using a combination of reflective capabilities in C# and Microsoft common intermediate language (CIL), and TRAP/BPEL[34] by wrapping the invocations and forwarding them to a local generic proxy developed in Java, respectively. To support CORBA programs developed using C++ ORBs, we plan to develop ACT/C++. We are also investigating techniques to support the insertion of hooks for adaptation into the operating system kernel,[19] the third case mentioned earlier.

Transparent shaping complements other work in adaptive software, particularly adaptive middleware. Figure 22 depicts this relationship,

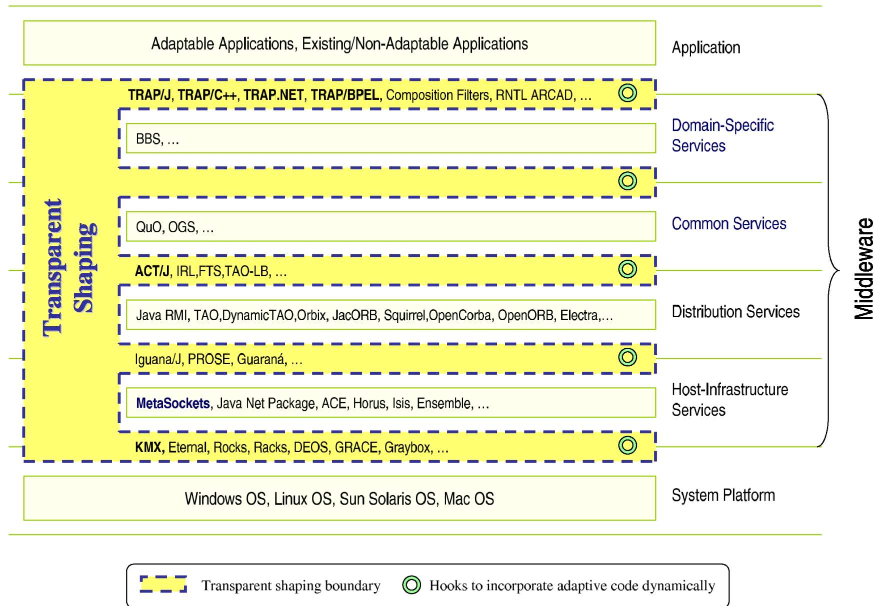

Fig. 22. Relationship of transparent shaping to other contributions.

according to Schmidt's four-layer middleware taxonomy.[35] Please note that the frameworks mentioned inside the transparent shaping boundary can be incorporated into existing applications transparently, while the ones outside this boundary require explicit calls from the application source code. As in our work with TRAP/J and MetaSockets, transparent shaping can enable existing non-adaptive applications to take advantage of adaptive host-infrastructure middleware services such as MetaSockets. Also, using our ACT/J framework, transparent shaping can enable existing CORBA applications to take advantage of adaptive common middleware services such as QuO. In addition, we note that many adaptive frameworks developed by other groups can be used to support transparent shaping. Examples include Composition Filters,[36] RNTL ARCAD,[10] Interoperable Replication Logic,[37] FTS,[38] TAO Load Balancing,[39] Iguana/J,[40] Prose,[41] Guaranà,[42] Eternal,[43] and Rocks/Racks.[44] Previously, we provided a summary of these and several other techniques.[1]

Finally, we note that transparent shaping has potential impact beyond supporting adaptation in individual programs, for example, to support application integration.[45] To integrate two existing heterogeneous

applications, possibly developed in different programming languages and targeted to run on different platforms, one needs to convert data and commands between the two applications on an ongoing basis. Transparent shaping offers a solution to this problem, without the need to modify application source code directly. In preliminary studies,[45,46] we have proposed several alternative architectures and showed how transparent shaping can support interoperability, via Web services, for Java RMI, CORBA, and .NET applications. As a proof of concept, we have conducted a case study that demonstrates the use of transparent shaping in the integration of an image retrieval application developed in CORBA with a frame grabber application developed in .NET.

7. Conclusions and Future Work

Transparent shaping supports reuse of existing programs in new, dynamic environments even though the specific characteristics of such new environments were not necessarily anticipated during the original design of the programs. In particular, many existing programs, not designed to be adaptable, are being ported to dynamic wireless environments, or hardened in other ways to support pervasive and autonomic computing. We have described an approach to transparent shaping based on the concept of program families and demonstrated how automated methods can be used to transform a program into another member of the same family. Our approach integrates four key technologies: aspect-oriented programming, behavioral reflection, component-based programming, and adaptive middleware. We highlighted two different realizations of transparent shaping, ACT and TRAP, and showed how they realize the general adaptive programming model. In addition to our work on other realizations of transparent shaping, as well as application integration, we are also addressing several other aspects of transparent shaping: coordination of adaptive behavior across system layers and among different systems, formal techniques to ensure that adaptations leave the system in a consistent state,[47] preventing adaptation mechanisms from being exploited by would-be attackers, and constructing "product lines" of adaptable software.

Acknowledgements

We express our gratitude to the faculty and students in the Software Engineering and Network Systems Laboratory at Michigan State University

for their feedback and their insightful discussions on this work. This work was supported in part by the US Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation Grants CCR-9912407, EIA-0000433, EIA-0130724, ITR-0313142, and CCR-9901017.

References

1. McKinley, P.K., Sadjadi, M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software, *IEEE Computer*, (July, 2004), pp. 56–64.
2. Weiser, M.: Ubiquitous computing, *IEEE Computer*. 26 (10), (October, 1993), pp. 71–72. ISSN 0018–9162.
3. Kephart, J.O., Chess, D.M.: The vision of autonomic computing, *IEEE Computer*, 36(1), 2003, pp. 41–50, ISSN 0018–9162.
4. Parnas, D.L.: On the design and development of program families, *IEEE Transactions on Software Engineering* (March. 1976).
5. Tarr, P., Ossher, H.: Eds. *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (W17)* (May, 2001).
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241 (June, 1997).
7. *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, (October, 2001), Vol. 44.
8. Maes, P.: Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, ACM Press (December, 1987), pp. 147–155, ISBN 0-89791-247-0.
9. Kiczales, G., des Rivières, J., Bobrow, D.G.: *The Art of Metaobject Protocols*. (MIT Press, 1991).
10. David, P.C., Ledoux, T., Bouraqadi-Saadani, N.M.N.: Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa (October, 2001).
11. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, (Addison-Wesley, 1999).
12. Bakken, D.E.: *Middleware* (Kluwer Academic Press, 2001).
13. *Proceedings of the Middleware'2000 Workshop on Reflective Middleware (RM2000)*, New York (April, 2000).
14. Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K., Sowell, J., Sadjadi, S.M., McKinley, P.K.: An aspect-oriented approach to dynamic adaptation. In *Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02)* (November, 2002).
15. *The Common Object Request Broker: Architecture and Specification Version 3.0*. Object Management Group, Framingham, Massachusett (July, 2003).
16. Cohen, G.A., Chase, J.S., Kaminsky, D.: Automatic program transformation with JOIE. In *1998 Usenix Technical Conference* (June, 1998).

17. Adve, S., Harris, A., Hughes, C., Jones, D., Kravets, R., Nahrstedt, K., Sachs, D., Sasanka, R., Srinivasan, J., and Yuan, W.: The Illinois GRACE project: Global resource adaptation through cooperation, 2002.
18. Distributed extensible open systems (the DEOS project), (2004). Georgia Institute of Technology — College of Computing.
19. Samimi, F., McKinley, P.K., Sadjadi, S.M., Ge, P.: Kernel-middleware cooperation in support of adaptive mobile computing. In *the Second International Workshop on Middleware for Pervasive and Ad-Hoc Computing*.
20. Sadjadi, S.M., McKinley, P.K.: ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan (March, 2004).
21. Sadjadi, S.M., McKinley, P.K.: Transparent self-optimization in existing CORBA applications. In *Proc. of the International Conference on Autonomic Computing (ICAC-04)*, (May, 2004), pp. 88–95, New York, NY.
22. *ORBacus for C++ and Java version 4.1.0.* IONA Technologies Inc., (2001).
23. Brose, G., Noffke, N.: JacORB 1.4 documentation. Technical report, Freie Universitt Berlin and Xtradyne Technologies AG (August, 2002).
24. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker, *Computer Communications*. 21(4) (April, 1998), pp. 294–324.
25. Sadjadi, S.M., McKinley, P.K.: Supporting transparent and generic adaptation in pervasive computing environments. Technical Report MSU-CSE-03-32, Department of Computer Science, Michigan State University, East Lansing, Michigan (November, 2003).
26. Zinky, J., Loyall, J., Shapiro, R.: Runtime performance modeling and measurement of adaptive distributed object applications. In *Proceedings of the International Symposium on Distributed Object and Applications (DOA 2002)*, Irvine, California (October, 2002).
27. Sadjadi, S.M., McKinley, P.K., Cheng, B.H.C., Stirewalt, R.K.: TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus (October, 2004).
28. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ, *Lecture Notes in Computer Science*, 2072, 2001, pp. 327–355.
29. Piveta, E.K., and Zancanella, L.C.: Aspect weaving strategies, *Journal of Universal Computer Science*, 9(8), 2003, pp. 970–983.
30. Rizzo, L., and Vicisano, L., RMDP: An FEC-based reliable multicast protocol for wireless environments, *ACM Mobile Computer and Communication Review*. 2(2) (April, 1998).
31. Fleming, S.D., Cheng, B.H.C., Stirewalt, R.K., and McKinley, P.K.: An approach to implementing dynamic adaptation in C++. In *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05), in conjunction with ICSE 2005*, St. Louis, Missouri (May, 2005).

32. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture, *Lecture Notes in Computer Science*, 707, 1993.
33. Sadjadi, S.M., Trigoso, F.: TRAP.NET: A realization of transparent shaping in .net. In *Proceedings of The Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2007)*, pp. 19–24, Boston, USA (July, 2007).
34. Ezenwoye, O., Sadjadi, S.M.: TRAP/BPEL: A framework for dynamic adaptation of composite services. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, Barcelona, Spain (March, 2007).
35. Schmidt, D.C.: Middleware for real-time and embedded systems, *Communications of the ACM*, 45(6) (June, 2002).
36. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters, *Communications of ACM*. (10) (October, 2001), pp. 51–57.
37. Baldoni, R., Marchetti, C., Termini, A.: Active software replication through a three-tier approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pp. 109–118, Osaka, Japan (October, 2002).
38. Hadad, E.: *Architectures for Fault-Tolerant Object-Oriented Middleware Services*. PhD thesis, Computer Science Department, The Technion — Israel Institute of Technology, 2001.
39. Othman, O.: The design, optimization, and performance of an adaptive middleware load balancing service. Master's thesis, University of California, Irvine, 2002.
40. Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (June, 2002).
41. Popovici, A., Gross, T., Alonso, G.: Dynamic homogeneous AOP with PROSE. Technical Report, Department of Computer Science, Federal Institute of Technology, Zurich, 2001.
42. Oliva, A., Buzato, L.E.: The implementation of Guaraná on Java. Technical Report IC-98-32, Universidade Estadual de Campinas (September, 1998).
43. Moser, L., Melliar-Smith, P., Narasimhan, P., Tewksbury, L., Kalogeraki, V.: The Eternal system: An architecture for enterprise applications. In *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)* (July, 1999).
44. Zandy, V.C., Miller, B.P.: Reliable network connections. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking* (September, 2002), pp. 95–106.
45. Sadjadi, S.M.: *Transparent Shaping for Existing Software to Support Pervasive and Autonomic Computing*. Ph.D. thesis, Department of Computer Science, Michigan State University, East Lansing, United States (August, 2004).
46. Sadjadi, S.M., McKinley, P.K.: Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the*

International Conference on Autonomic Computing (ICAC'05), Seattle, Washington (June, 2005).

47. Zhang, J., Yang, Z., Cheng, B.H.C., McKinley, P.K.: Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland (May, 2004).