Computer Networks 53 (2009) 1570–1586

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

# Transparent autonomization in CORBA

S. Masoud Sadjadi [a,*], Philip K. McKinley [b]

[a] Florida International University, Miami, FL 33199, United States
[b] Michigan State University, East Lansing, MI 48824, United States

ARTICLE INFO

*Article history:*
Available online 29 December 2008

*Keywords:*
Transparent shaping
Adaptive middleware
CORBA
Autonomic computing
Self-optimization
Dynamic adaptation
Quality-of-service
Mobile computing
Generic proxy

ABSTRACT

Increasingly, software systems are constructed by integrating and composing multiple existing applications. The resulting complexity increases the need for self-management of the system. However, adding autonomic behavior to composite systems is difficult, especially when the constituent components are heterogeneous and they were not originally designed to support such interactions. Moreover, entangling the code for self-management with the code for the business logic of the original applications may actually increase the complexity of the systems, counter to the desired goal. In this paper, we address autonomization of composite systems that use CORBA, one of the first widely used middleware platforms introduced more than 17 years ago that is still commonly used in numerous systems. We propose a model, called Adaptive CORBA Template (ACT), that enables autonomic behavior to be added to CORBA applications automatically and *transparently*, that is, without requiring any modifications to the code implementing the business logic of the original applications. To do so, ACT uses "generic" interceptors, which are added to CORBA applications at startup time and enable autonomic behavior to be introduced later at runtime. We have developed ACT/J, a prototype of ACT in Java. We describe a case study in which ACT/J is used to introduce three types of autonomic behavior (self-healing, self-optimization, and self-configuration) to a distributed surveillance application.

Published by Elsevier B.V.

## 1. Introduction

Driven by the Internet revolution and its effects on information technology, the last decade has witnessed proliferation of integration middleware technologies addressing software integration problems [1]. Instead of developing new software systems from scratch, the focus of integration middleware technologies has been on leveraging available software resources by enabling their interoperation [2]. CORBA 2.0 [3], released in 1996, was among the first middleware technologies to address integration issues [4], and since then CORBA has been successfully used in the integration of numerous software systems [5].

A typical CORBA application comprises heterogeneous software components, often developed in different programming languages and targeting different platforms (operating systems, devices, and networks). Indeed, a major goal of CORBA and other middleware platforms is to hide this heterogeneity from the business logic of the integrated applications. While this approach helps developers to integrate their systems more easily, the management of complex CORBA-based systems is challenging, especially as they evolve to accommodate new software and hardware technologies. In particular, managing composite systems involves ensuring non-functional concerns such as quality-of-service, fault tolerance, and security. Unfortunately, these concerns are often directly affected by the underlying technologies and the environments in which the application is deployed.

*Autonomic computing* [6] promises a general solution to the management problem that relies on complex systems

* Corresponding author. Tel.: +1 305 348 1835.
*E-mail addresses:* sadjadi@cs.fiu.edu (S.M. Sadjadi), mckinley@cse.msu.edu (P.K. McKinley).
*URLs:* http://www.cs.fiu.edu/~sadjadi (S.M. Sadjadi), http://www.cse.msu.edu/~mckinley (P.K. McKinley).

1389-1286/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.comnet.2008.12.012

to manage themselves. Instead of requiring low-level interaction with users or system administrators, self-managing systems would require only high-level human guidance – defined by goals and policies – in order to work as expected. Each autonomic element in the system comprises a *managed element*, implementing the business logic of the system, and an *autonomic manager*, implementing the self-managing behavior of the system. However, self-management concerns (self-healing, self-optimization, self-configuration, and self-protection) tend to crosscut the functional decomposition in the managed elements [7–9]. Consequently, if the code for self-management is entangled with the code for the business logic of the original systems, then the complexity of managing the resulted autonomic system may actually increase, contradicting the purpose of autonomic computing.

This paper describes the *Adaptive CORBA Template* (*ACT*), a framework that enables dynamic addition of autonomic behavior to existing CORBA systems, without modifying the application code. At startup time, ACT turns the constituent software programs into managed elements by transparently inserting generic hooks capable of intercepting all CORBA remote interactions. Next, at run time, these hooks can be used to introduce autonomic managers into the system. An autonomic manager in turn can intercept the requests, replies, and exceptions that pass through the CORBA Core (called ORB, which stands for Object Request Broker), adapting or redirecting them as needed. Effectively, ACT enables *transparent autonomization* (i.e., transparent addition of self-managing behavior) in CORBA applications.

We identify three types of applications that may benefit from such a capability. First, dependable applications are required to operate continuously without interruption; code for handling newly discovered faults and in general self-managing behavior can be added to these applications as they execute. Second, embedded applications are required to provide very small footprints; a minimal autonomic code can be added to the application at compile or startup time, while optional and temporary autonomic code can be swapped in and out as needed during run time. Third, the source code for some legacy CORBA applications may be unavailable, or modifying the source code may be undesirable. Such applications can be autonomized transparently using ACT, without modifying or even recompiling the original application source code.

Various aspects of the ACT framework have been described in earlier conference papers [10,11]; this paper provides a complete picture of the ACT project, presents a more comprehensive architectural solution, and includes additional details of the autonomization process and experimental results for three different types of autonomic behavior: self-healing, self-optimization, and self-configuration. The remainder of this paper is organized as follows. Section 2 provides a background on CORBA and describes the architecture and operation of ACT, as well as a Java prototype, ACT/J. Section 3 presents a case study where we used the ACT prototype to add three different types of autonomic behavior to an existing surveillance application. Section 4 categorizes related research projects, and Section 5 provides concluding remarks.

## 2. ACT architecture and operation

ACT is intended to support the construction of adaptive CORBA applications from existing CORBA applications transparently, that is, without modifying the original application functionality. ACT should enable CORBA applications to support adaptive behavior at run time without the need to stop, modify, recompile, relink, or restart the applications. Moreover, ACT should introduce nominal overhead to the performance of the existing applications. With these design goals in mind, we developed a two-step process that supports transparent autonomization in existing CORBA applications. In the rest of this section, we provide a brief overview of CORBA and the autonomization process, describe the architecture and internal operation of ACT, and discuss the prototype implementation of ACT in Java.

### 2.1. CORBA background

The *Common Object Request Broker Architecture* (*CORBA*) [12] is an integration and distribution middleware specification defined by the Object Management Group (OMG) [4]. Fig. 1 depicts a simple client–server CORBA application comprising a client and a server program and their orientation among three system layers: application, middleware, and network.

Let us assume that the client has a valid reference to the CORBA object realized by the servant. For clarity, a broker program is not shown. The *Object Request Broker (ORB)*, the core of CORBA, allows objects to interact transparently with other objects (located locally or remotely). A CORBA object is represented by its interface, is identified by its reference, and is realized in an object-oriented program as a local object called the *servant*. The client calls methods on the servant as if the CORBA object were located in the client address space. The *Interface Definition Language (IDL)* is a language for defining CORBA interfaces. An IDL compiler is used to automatically generate the code for stubs and skeletons. An *IDL stub* represents a servant locally in the client address space and an *IDL skeleton* represents a client locally in the servant address space. IDL stubs and skeletons marshal and unmarshal requests and responses to enable object interactions over a network.

*CORBA Portable Request Interceptors* provide a transparent mechanism to intercept messages (reified requests, replies, and exceptions) inside the ORBs of a CORBA application. For example, a portable interceptor can be used to forward a particular request to a *different* CORBA object (e.g., forwarded request flows 2, 3, and 4 in Fig. 1). However, to ensure portability, interceptors are not allowed to reply to intercepted requests or to modify the parameters [12]. This restriction limits the ability of request interceptors alone to adapt the behavior of CORBA applications.

### 2.2. Autonomization process

Fig. 2 illustrates the two-step process to autonomize an existing CORBA application using ACT. As this process is

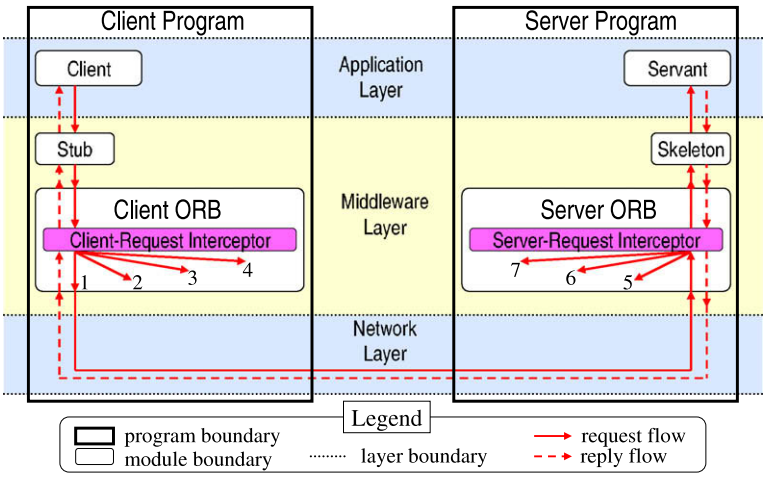


**Fig. 1.** CORBA request portable interceptors.

similar for both server- and client-side programs, in this figure we only show the client-side process. Also, for clarity, some details such as stubs, skeletons, and IIOP (Internet Inter-ORB Protocol) are not shown. During the first step, which occurs at startup time, a *generic interceptor* is added to the existing CORBA application (`Client GI` for the client program and `Server GI` for the server program). A generic interceptor is a specialized request interceptor that is registered with the ORB of a CORBA application at startup time, but enables late (dynamic) registration of other portable interceptors. Once registered, a generic interceptor intercepts all the requests, replies, and exceptions passing through the ORB. The initial behavior of a generic interceptor is simply to forward all the intercepted interactions to their original destinations. Therefore, when there is no need for dynamic adaptation, the original application functionality and behavior is preserved. As we will show in later sections, the overhead of the interception and redirection by a generic interceptor is negligible for most applications. At this point, the CORBA program (with the generic interceptors registered both at the client- and server-side programs) is called a *managed element*, or an *adapt-ready program*.

As illustrated in Fig. 2, during the second step, which occurs at run time, the generic interceptors in the adapt-ready CORBA programs can be controlled remotely (*e.g.*, by using administration consoles supported in ACT) to load the ACT Core (`Client ACT Core` for the client program and `Server ACT Core` for the server program). The right side of Fig. 2 shows how the client generic interceptor redirects the flow of a request/reply to the client ACT core. The ACT core basically encapsulates the code that responds to the unanticipated changes at run time. In the autonomic computing terminology, this code is known as "autonomic manager" [6]. Once such code is added to an adapt-ready program, the application is called an *autonomic element*, or an *ACT-ready program*. For flexibility, making the program to be an autonomic element can be done either at startup time (right after the program has become adapt-ready) or it can wait until run time when the need arises.

### 2.3. ACT architecture

ACT comprises two main types of components: generic interceptors and instances of the ACT core. As depicted in Fig. 3, the *client* generic interceptor intercepts all outgoing

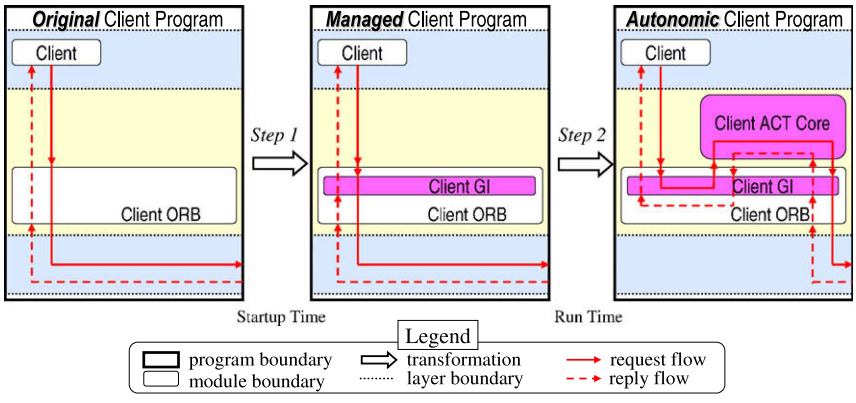


**Fig. 2.** ACT autonomization steps.

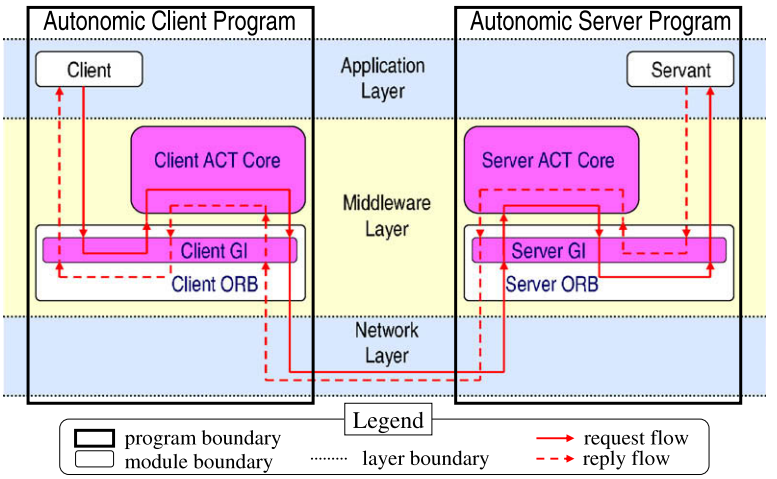


**Fig. 3.** ACT configuration in the context of a simple CORBA application.

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.

Fig. 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.

*2.3.1. Dynamic interceptors*

According to the CORBA specification [12], 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 when they are no longer needed. 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 the 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

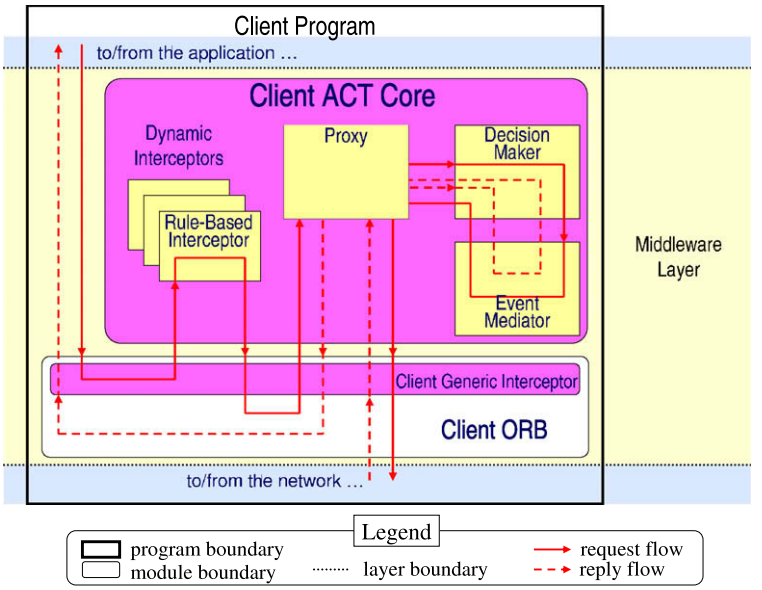


**Fig. 4.** ACT core components interacting with the rest of the system.

rule matches a request, a rule-based interceptor consults the rule's condition object. If 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 [12]. 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*.

#### 2.3.2. Proxies

A proxy is a special CORBA object that plays the role of a surrogate for other CORBA objects. A proxy receives requests originally targeted for its corresponding CORBA object, as depicted in Fig. 4. 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).

In ACT, there are two types of proxies: specific and generic proxies. A *specific proxy* is a surrogate for a specific CORBA object. It provides the same set of methods as the CORBA object (that is, a specific proxy is required to implement the same interface as that of its corresponding CORBA object). A *generic proxy*, in contrast, is a surrogate for *all* CORBA objects. A generic proxy can receive requests destined to any CORBA object and can send requests to the original target CORBA objects or to any other CORBA objects dynamically. As a result, a generic proxy may introduce more overhead than a specific proxy. A generic proxy is generated per language, whereas a specific proxy is generated per CORBA object. Due to its special role in ACT, the generic proxy is described in more detail in a dedicated subsection (see Section 2.5).

#### 2.3.3. 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.

#### 2.3.4. Event mediators

*An event mediator* is a CORBA object that decouples event generators from event listeners using a publish/subscribe approach. We adopted this concept from the work by Bacon et al. [13]. An event mediator publishes a listener service, enabling registration of CORBA objects as event listeners. The event mediator is informed of events through a notification service. An event mediator forwards a copy of a new event to all listeners that have registered interest in this type of event.

### 2.4. ACT Internal operation

To describe the interactions among the ACT components, we refer again to the configuration shown in Figs. 3 and 4. Here, we consider only the activities on the client side and, for clarity, stubs and skeletons are not considered.

First, the request from the client to the servant is forwarded to the proxy. After the request is received by the client ORB, it is intercepted by the client generic interceptor, where it is forwarded to the client rule-based interceptor. The client rule-based interceptor checks its active rules. In this scenario, let us assume it finds a rule that matches the request. The rule raises a ForwardRequest exception, which is passed to the client generic interceptor and then to the client ORB, where the request target is changed to the proxy. Before the new request is sent to the proxy, it is intercepted again by the client generic and rule-based interceptors, but this time no exception is raised, and the call simply returns. The proxy receives the request.

Next, the proxy processes the request and forwards it to the servant. The proxy consults the decision maker, where an event may be raised to handle an unexpected situation. The decision maker may adapt the client application by modifying the request parameters, sending new requests to other objects, or directing the proxy to reply to the request (e.g., using cached replies). We assume that in this scenario, the decision maker modifies the request parameters and directs the proxy to send the modified request to the servant via the client ORB. The modified request is also intercepted by the client generic and rule-based interceptors, but again no exception is raised. Therefore, the modified request is sent to the server ORB.

The reverse sequence of actions occurs at the server application (not shown), and the reply to the modified request is returned to the client ORB. The reply is intercepted by the client generic interceptor and rule-based interceptors, where no exception is raised. The reply is sent back to the proxy, where it is forwarded to the decision maker for possible modifications and possible event raising.

Finally, using the reply from the servant and the direction given by the decision maker, the proxy replies to the client's request. The reply is intercepted by the client generic and rule-based interceptors. Again no exception is raised, and the client ORB sends the reply back to the client.

The extensive redirecting of messages in ACT raises the issue of performance overhead. We deem such overhead as necessary to provide flexibility and transparency. Moreover, our experimental results, described in Section 3.2, indicate that the overhead is actually quite small.

### 2.5. Generic proxy

To enable dynamic weaving of adaptive functionality that is common to multiple applications, 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) of specific applications.

As opposed to a *specific proxy*, which provides the same set of methods as the target CORBA object, the *generic proxy* is a particular CORBA object that is able to receive *any* CORBA request. To determine how to handle a

particular request, the generic proxy accesses the CORBA interface repository [12], 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 allows the user to introduce the interface repository to the application ORB. Providing IDL information to the generic proxy via the interface repository 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, which are typically available.

In its default operation, 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*.

Fig. 5 illustrates the sequence of a request/reply in 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 generic 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), the generic proxy will continue its operation by invoking the request. If the reply to the request is returned by the decision maker, the proxy replies to the original request using the reply from the decision maker. The generic proxy uses the CORBA dynamic skeleton interface (DSI) [12] to receive any type of request. The generic proxy and the rule-based decision maker use the CORBA dynamic invocation interface (DII) [12] to create and invoke a new request dynamically.

### 2.6. ACT/J: ACT prototype in Java

We have developed ACT/J, a prototype of ACT in Java, and used it to evaluate the ACT framework in real applications. We tested ACT/J atop ORBacus [14], a CORBA-compliant ORB distributed by IONA Technologies. ORBacus, like JacORB [15], TAO [16], and many other CORBA ORBs, supports CORBA portable interceptors, the only requirement for using ACT.

To make a CORBA application ACT-ready at 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,

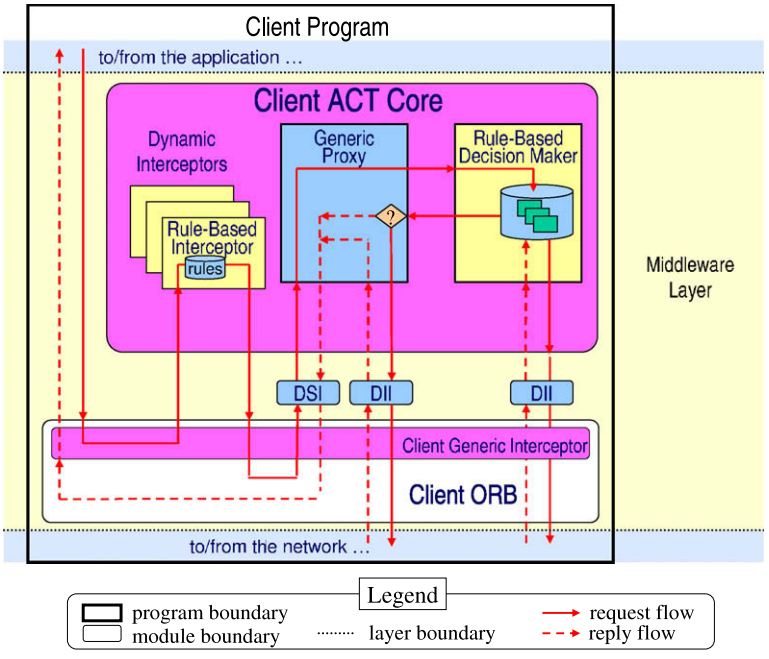


**Fig. 5.** Incorporating generic proxy in the ACT core.

although the CORBA specification does not support this feature. To implement ACT/J over an ORB that does not provide such a reference, we could simply create a new ORB, although its use introduces additional overhead.

In this study, the composer of the adaptive application is assumed to be a human, who performs dynamic adaptation using the administrative consoles. Therefore, to test the operation of ACT/J, we developed two administrative consoles: the Interceptor Registration Console and the Rule Management Console. In other settings, the composer might be a piece of software – an aspect weaver, a component loader, a runtime system, or a metaobject [7]. 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. In the next section, we describe a case study in which we use ACT/J to enhance an existing CORBA application with different types of autonomic behavior.

## 3. Case study: an autonomic surveillance application

To evaluate ACT/J, we conducted a case study in which self-management code is woven into an existing CORBA application, without modifying the application source code. The experiments demonstrate that ACT/J is capable of adding self-healing, self-optimization, and self-configuration to CORBA applications with negligible performance overhead. We begin this section with a brief overview of the existing application, followed by the description of the experiments.

### 3.1. The example application

Surveillance systems are becoming an important part of our daily life. Such systems are commonly deployed in airports, banks, offices, and even individual homes. Currently, surveillance systems are undergoing a transition from traditional analog solutions to digital ones. Compared to traditional analog systems, digital surveillance systems offer better flexibility in video/image content processing, transmission, motion detection, facial recognition, and object tracking.

Fig. 6 shows an example surveillance system. The left side of the figure shows the physical configuration; the system comprises two types of components: video/image capture and control components. The capture component usually includes one or more cameras and an encoder device. This component captures the raw video or images and compresses the data using a prescribed coding standard (*e.g.,* MPEG2, MPEG4, or H.263). The video control component monitors the video channels and controls the operation of the cameras. The right side of Fig. 6 depicts our strategy for making each such component an autonomic element. The autonomic manager is the ACT core and the redirection (interception and modification arrows) is done by the generic interceptor.

In the case study, we used an existing and freely available CORBA image-retrieval application developed previously by BBN Technologies [17]. As illustrated in Fig. 7, this application has two parts: a client program (image client) that continuously retrieves images from a server program (described next) and displays the images on the screen as soon as they are available; and a server program (image server) that stores images, wait for the requests for images, and replies to each request by sending back an image. The image server provides four different versions of each image, varying in size and quality. Typical comparative file sizes we used in the experiments are 90, 25, 14, and 4 KB, corresponding to large-processed, large-unprocessed, small-processed, and small-unprocessed.

The image-retrieval application was developed in Java and was distributed as part of the *Quality Objects* (QuO) framework. QuO [18] is released under an open-source license. It is a powerful adaptive framework that supports dynamic adaptation in CORBA and Java RMI applications. Some background on QuO is provided later in this section. Please note that except for the last experiment described in this section, we did not use the native adaptability features in QuO. Rather, our focus in on how ACT/J can be used to enhance the application without modifying the application code.

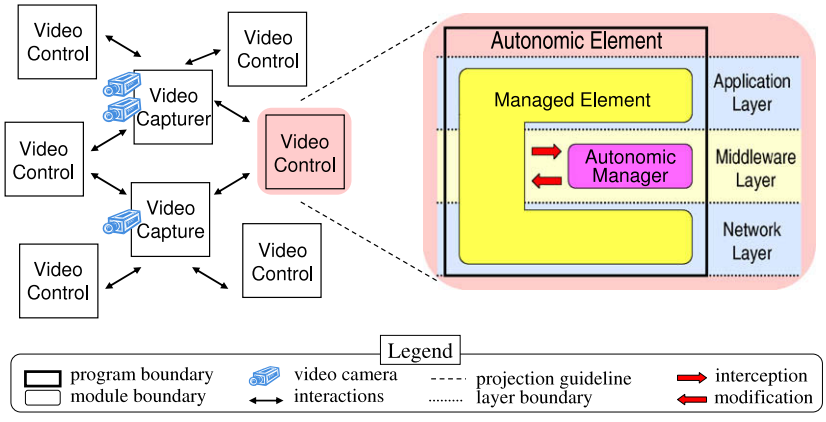


**Fig. 6.** The architecture of an autonomic video surveillance application.

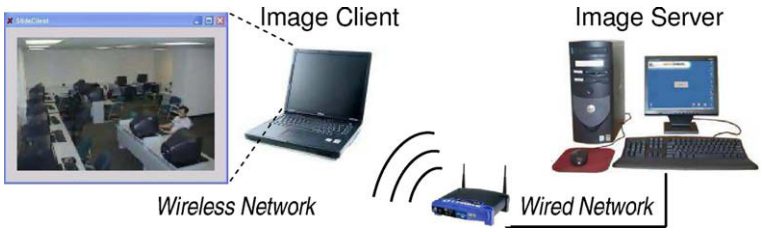


**Fig. 7.** The image-retrieval application developed previously by BBN Technologies [18]: Recommended configuration: Linux OS and wired network.

### 3.2. ACT/J overhead

We autonomized the image-retrieval application in two steps. First, we made the application adapt-ready by introducing a generic interceptor to the image client and image server at startup time. To do so, we started the image client and server applications with a command-line parameter directing them to an ORBacus configuration file defining how to load, create and register a generic interceptor with their respective ORBs. Next, using the generic interceptors and the administrative consoles, we inserted the autonomic manager (the ACT core) in both image client and image server, converting them to become autonomic elements (ACT-ready programs). Since the default behavior of ACT is to intercept all the remote interactions and forward them to their original targets, this configuration enables us to measure the overhead of introducing this control mechanism.

To evaluate the overhead of using ACT/J, we conducted two sets of experiments: for one set, we used the original image-retrieval application; and for the other set, we used the autonomic version of the application. For each set, we used images of varying size (14 different sizes ranging from 1 KB to 8 MB) and evaluated the average round-trip delay on a low traffic 11 Mbps wireless network. The image server was running on a desktop computer connected to an 802.11b wireless access point through a wired network, and the image client was running on a laptop computer connected to the access point through the wireless network. The laptop was kept stationary throughout this experiment. Fig. 8 compares the round-trip delay for retrieving images of varying size, using both the original application and the autonomic version. As shown, this overhead introduced by ACT/J is negligible.

### 3.3. Self healing

In the next set of experiments, the client code executes on the laptop of a mobile user who is monitoring a physical facility through continuous still images drawn from multiple camera sources. We executed the server on a desktop computer connected to a 100 Mbps wired network. Both the desktop and laptop systems are running the Linux operating system.

The client laptop is located in a three-cell wireless network, and the objective of this exercise is to introduce hand-off behavior to the application enabling its connectivity to be switched among different wireless access points without interrupting the application. Fig. 9 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

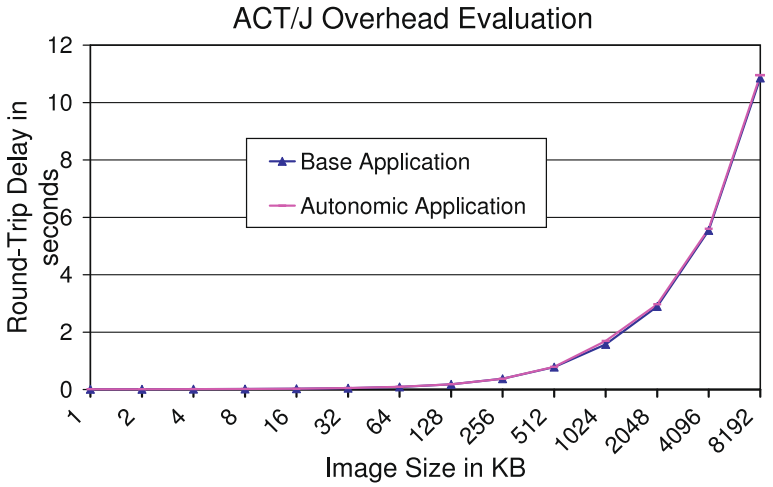


**Fig. 8.** Overhead of using ACT/J measured by round-trip delay in request/reply.

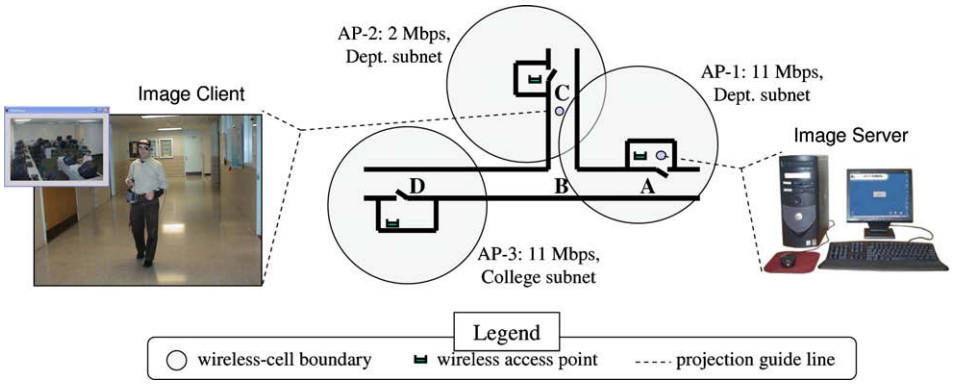


**Fig. 9.** The configuration of the access points used in the experiment.

managed by our Computer Science and Engineering (CSE) 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 self-healing rules in ACT are aware of such handoff in mobile computing and as such they provide a better connectivity for the application.

Fig. 10 shows two plots corresponding to the two experiments we conducted to demonstrate the self-healing behavior of the autonomized surveillance application: one plot corresponds to the experiment with the original application and the other plot corresponds to the experiment with the autonomized application. In both experiments, for the first 120 s, the user stays close to the location A and then at time 120 s the user starts walking from location A to location B for 60 s (see Fig. 9 for location reference). As illustrated in Fig. 10, for the first 180 seconds into the experiments, the frame rates for the two experiments remains at about 4.5 frames per second (the frame rate for the autonomized application is a bit less than the original application because of the overhead introduced by ACT). At time 180 s, the user begins walking from B to C and then back from C to B for 120 s. For this period, as the subnetwork does not change and the IP address provided by the Department DHCP server is still valid, both applications switch to the access point at location C (AP-2) without any problem. During this period, the frame rate goes down to 2 frames per second because of the lower bandwidth of AP-2 (2 Mbps vs. the 11 Mbps bandwidth of AP-1).

At point 300 s, the user begins walking from B to D and then back from D to B for 180 s. During this period, the original application loses connection and the frame rate goes to zero. Even when the signal for AP-3 is strong enough, the original application still cannot use this wireless network to resume its operation because the IP address assigned by the Department is not valid for the College network. The autonomized application, however, tries aggressively to maintain the connection. It can work also with the new IP address assigned by the College network and for a significant part of this 180 s period, it maintains a high frame rate. Finally, at point 480 s, the user begins walking from B back to A for 120 s. During this period, both applications try to use AP-1 for their operation (the autonomized version is quicker in this regard) and they both

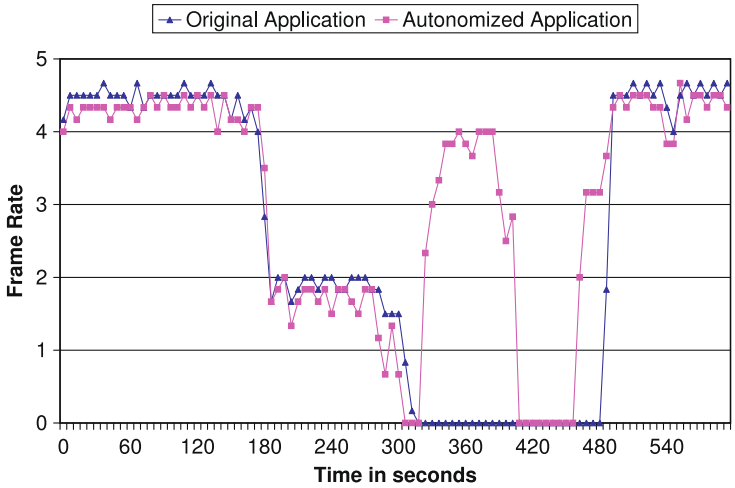


**Fig. 10.** Frame rate demonstrating the self-healing behavior added by ACT.

maintain a 4.5 frame rate for the rest of experiments. We note that if the period of disconnection is long enough, the original application will crash (a network exception is thrown by CORBA ORBs and the original application does not know how to handle this exception).

### 3.4. Self optimization

To investigate how ACT/J can support self-optimization, 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 to weave the new 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.

Fig. 11 shows the Automatic Adaptation Console, which diplays the application status and also enables the user to enter quality-of-service preferences. As shown in this figure, 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 a hand-off from one wireless cell to another one, 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 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.

Fig. 12 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. 11. For the first 60 s of the experiment, the user stays close to location A (Fig. 9). 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 ms to maintain the frame rate at about 2 frames per second. At time 120 s, the user starts walking from location A to location B for 60 s. The automatic adaptation rule maintains the frame rate by decreasing the inter-frame delay during this period.

At time 180 s, the user begins walking from location B to location C and back again, returning to location B at

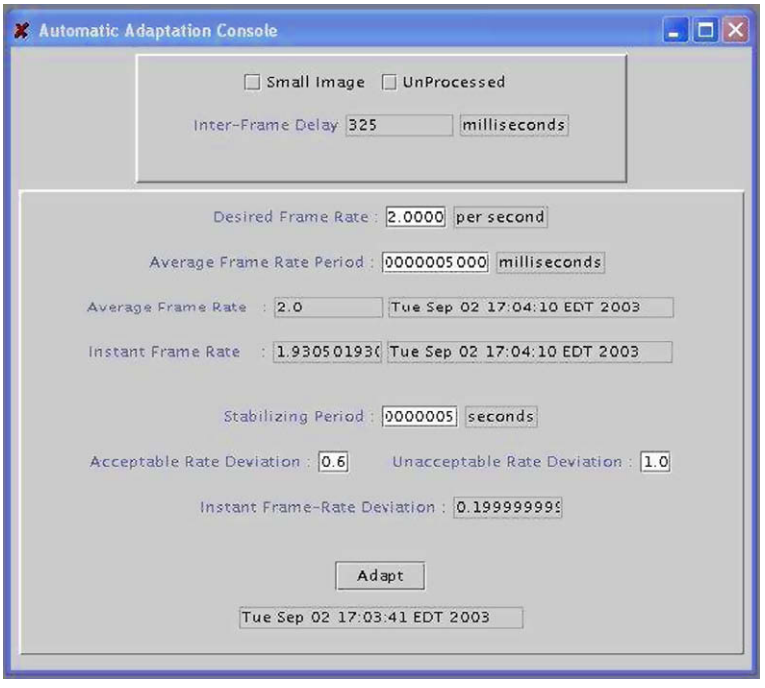


**Fig. 11.** Automatic Adaptation Console.

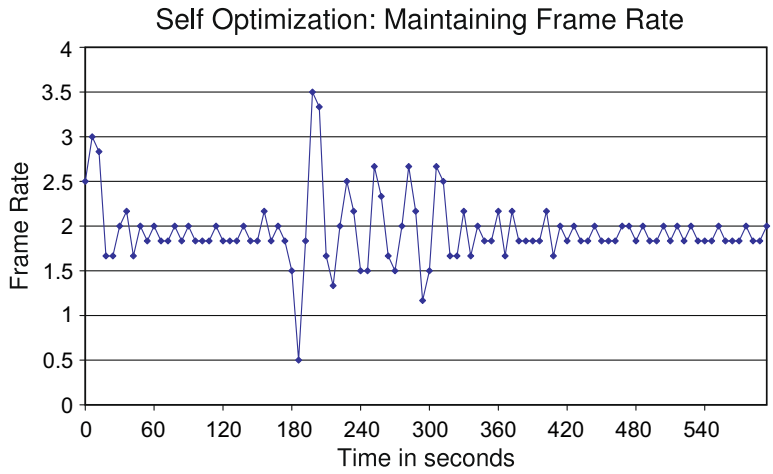


**Fig. 12.** Maintaining the application frame rate using automatic adaptation.

360 s. 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 ms to maintain the frame rate at 2 frames per second. Although there is some oscillation, the rate stabilizes by time 360 s. 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 that expected, an inter-frame delay of about 200 ms 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-optimizing behavior to a CORBA application transparently and dynamically.

### 3.5. Self configuration

To investigate the use of ACT in transparent self-configuration, we combined ACT/J with QuO. ACT and QuO can work together in two major ways. First, ACT enables legacy CORBA applications to incorporate and benefit from QuO functionality, without modifying the source code of the application (indeed, even if the source code is unavailable). Such a need may arise if the application is to be executed in an environment where conditions might be quite different than originally planned. Second, combining QuO and ACT enables weaving of adaptive code into distributed applications at both compile time and run time; we describe a specific example later in this section. We begin a brief overview of QuO, for completeness, followed by a discussion of how ACT and QuO interact and a description of an experiment in which they were combined to enhance an extant application.

#### 3.5.1. QuO background

QuO employs aspect-oriented programming [19] to separate the non-functional aspects from the functional aspects of an application. Fig. 13 illustrates a very simple QuO application. The client wrapper (or *delegate*) is the main point of contact between the client and the QuO core. The client wrapper is generated from a program written in the aspect-oriented structural description language (ASL) [20]. The QuO core comprises a contract and several system conditions. A *contract* is written in the contract-description language (CDL) [20] and defines acceptable regions of operation. *System conditions* can be considered as software "sensors" that record values representing the state of the execution environment. QuO combines the code for the QuO core and the code for wrapper into a package called a *qosket*. Using an aspect weaver called quogen [18], QuO weaves a qosket into an application at compile time.

As shown in Fig. 13, a request from the client is first received by the client wrapper. In a typical CORBA application, a client has a reference to a CORBA object stub. In QuO, however, the application developer explicitly creates the client wrapper, which wraps the stub (the wrapped stubs are not shown). The client wrapper consults the contract in the client QuO core. The contract evaluates the current acceptable region of operation according to the details of the request and the status of the system as monitored by the system-condition objects. Once the current region of operation is identified, the actions specified in the contract are carried out. These actions might include returning a cached reply to the client, sending a request different than the original, forwarding the request with modified parameters, or redirecting the request to another CORBA object. If the reply is not generated locally, the request (or a modified request) is passed to the client ORB. The request is then sent to the server side of the application, where the reverse sequence of actions occurs. The reply generated by the servant, possibly modified by the server QuO core, will eventually reach the client ORB, where it is passed to the client wrapper. The client wrapper consults the client QuO core again for possible modifications and, finally, returns the reply to the client.

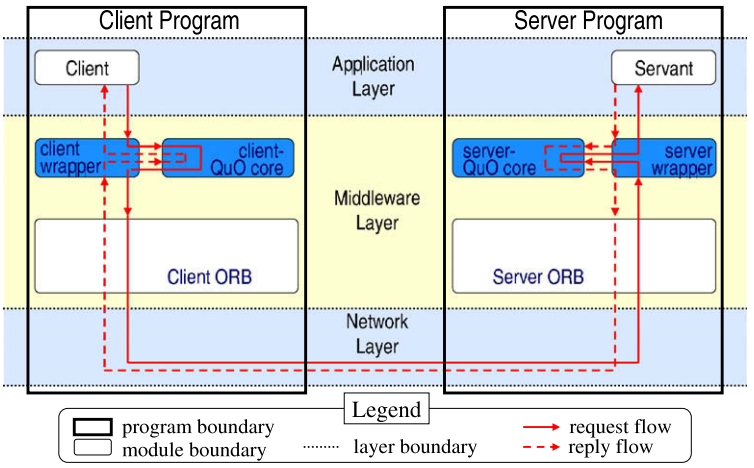


**Fig. 13.** A simplified depiction of the QuO architecture.

*3.5.2. Dynamic weaving of qoskets using ACT*

Combining ACT with QuO enables transparent weaving of new qoskets into applications at run time. Fig. 14 shows a request/reply sequence in a simple CORBA application using both QuO and ACT. The client and server generic interceptors are registered with the client and server ORBs, respectively, at startup time. To weave a new qosket into the application at run time, a new rule can be inserted in the client rule-based interceptor. The new rule can direct the rule-based interceptor to load the code for a proxy and a decision maker. The proxy in this case is simply a modified QuO wrapper, and the decision maker is exactly the contract defined in the new qosket. The rule then intercepts all incoming and outgoing requests/replies and forwards them to the proxy, where they are processed as if the qosket had been woven in to the application at compile time.

*3.5.3. Example: supporting unanticipated adaptation*

To evaluate the performance and functionality of the hybrid ACT/QuO architecture described above, we used it to insert new adaptive functionality into the image-retrieval application at run time. This application supports several different types of qoskets, which can be woven into the application at startup time. A particular qosket called "UserAdapt" enables a user to modify the application interactively by directing it to retrieve different versions of the images. For example, selecting small instead of large versions of images can be used to reduce bandwidth consumption and delay.

We developed a new qosket called UserAdaptFrameRate to weave to the application at run time using ACT/J. This qosket enables the user to interactively control the rate at which images are retrieved. The code that defines the contract (in CDL) for the new qosket is listed below:

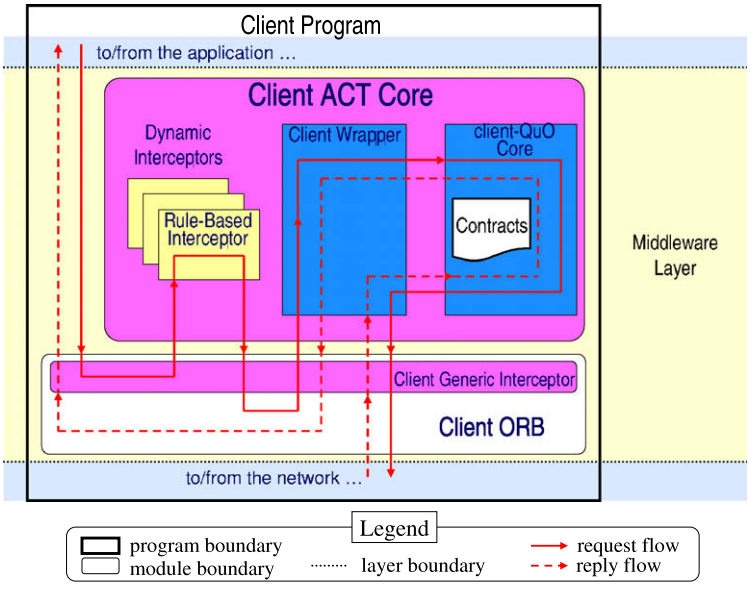


**Fig. 14.** Coupling ACT and QuO.

```
contract UserAdaptFrameRate (syscond
quo::ValueSC quo_sc::ValueSCImpl userFrameRate)
{
   region Fast (userFrameRate == 2) {}
   region Normal (userFrameRate == 1) {}
   region Slow (userFrameRate == 0) {}
};
```

The code that defines the wrapper (in ADL) for the new qosket is listed below:

```
behavior UserAdaptFrameRate ()
{
   void slide::SlideShow::read(in long gifNum-
   ber, out string size, out octetArray buf)
   {
      before METHODCALL
      {
         region Fast {}
         region Normal {... Thread.sleep(50);...
         }
         region Slow {... Thread.sleep(100);... }
      }

   }...
}
```

We defined three regions of operations Fast, Normal, and Slow in the contract, enabling the user to control the frame rate, for example, to conserve bandwidth. This control is accomplished by inserting appropriate delays. For the Fast region, we did not insert any delay, but for the Normal and Slow regions, we inserted 50 and 100 ms frame-interval delays, respectively. We used the quogen utility to compile the new qosket.

#### 3.5.4. Experimental results

To demonstrate the interaction between ACT and QuO, we ran an experiment that involves both static and dynamic weaving of qoskets into this application. The experiment is intended to represent run time upgrading of a surveillance system to add a new feature that controls the frame rate.

We executed the server on a desktop computer connected to a 100 Mbps wired network and the client on a laptop computer connected to an 11 Mbps 802.11b wireless network; both systems are running the Linux operating system. At startup time the "UserAdapt" qosket is woven into the application by specifying the wrapper class as a command-line parameter. Later, at run time, we used our Interceptor Registration Console to weave the "UserAdaptFrameRate" qosket into the application. Fig. 15 shows two screen dumps of the application: the top one displays the large version of an image and the bottom one displays the small version.

Fig. 16 shows a trace of the rate at which frames are displayed at the client application. During the experiment, a user modifies the application as follows. When the application starts, large versions of frames (the default option) are retrieved from the server as fast as possible. The size of these images, combined with the limited bandwidth of the wireless network, produces a frame rate of approximately 2 images per second for the first 30 s of this experiment. At this point, the user selects the small-images option by way of the GUI in the "UserAdapt" qosket, thereby increasing the frame rate to approximately 14 images per second.

At 60 seconds into the experiment, the user dynamically weaves the UserAdaptFrameRate qosket into the application, using the interactive administration utilities. Fig. 16 shows a short, downward spike in the frame rate caused by the delay for weaving the new qosket. We consider such a one-time delay to be acceptable for this type of application. Immediately after the qosket is inserted, an interactive console is displayed by the qosket, enabling the user to choose from the three options (Fast, Normal, and Slow) interactively at run time. The Fast option is the default. At 90 s into the experiment, the user selects the Normal option; the additional 50 ms delay reduces the frame rate to approximately 7.5 images per second. At 120 s, the user chooses the Slow option (100 ms delay), which reduces the frame rate to approximately 5.5 images per second. At 150 s, the user chooses the Fast option again, which increases the frame rate to 14 images per second.

This experiment illustrates how ACT can be used to dynamically incorporate new behavior (in this case, a new QuO qosket) into a CORBA application at run time. The process is transparent to the application, in that we did not modify the application code or the QuO code. We simply restarted the application and specified the ACT generic proxy in a command-line parameter.

## 4. Related work

We designed ACT to enable transparent autonomization to CORBA applications. However, the autonomic behavior might be provided by other adaptive frameworks. Specifically, ACT can be used to dynamically load components of one adaptive framework into an existing CORBA application that was developed using a different framework. By transparently intercepting requests and replies, ACT enables such applications to exploit adaptive functionality defined in other frameworks. We refer to such a system as a *framework gateway*. Next, we discuss several adaptive middleware frameworks and their relationship to ACT. We group the frameworks into three categories: aspect-oriented middleware, reflective middleware, and interception-based middleware.

### 4.1. Aspect-oriented middleware

Aspect-oriented middleware enables separation of the functional aspects from the non-functional aspects (e.g., quality-of-service, security, and fault-tolerance) of a distributed application [21,22,8,9]. One of the most extensive projects in this area is Quality Objects (QuO) [18,23], which provides an adaptable framework to support QoS in CORBA applications. QuO weaves QoS aspects, referred to as *qoskets*, into the applications at compile time by wrapping stubs and skeletons with specialized *delegates*, which

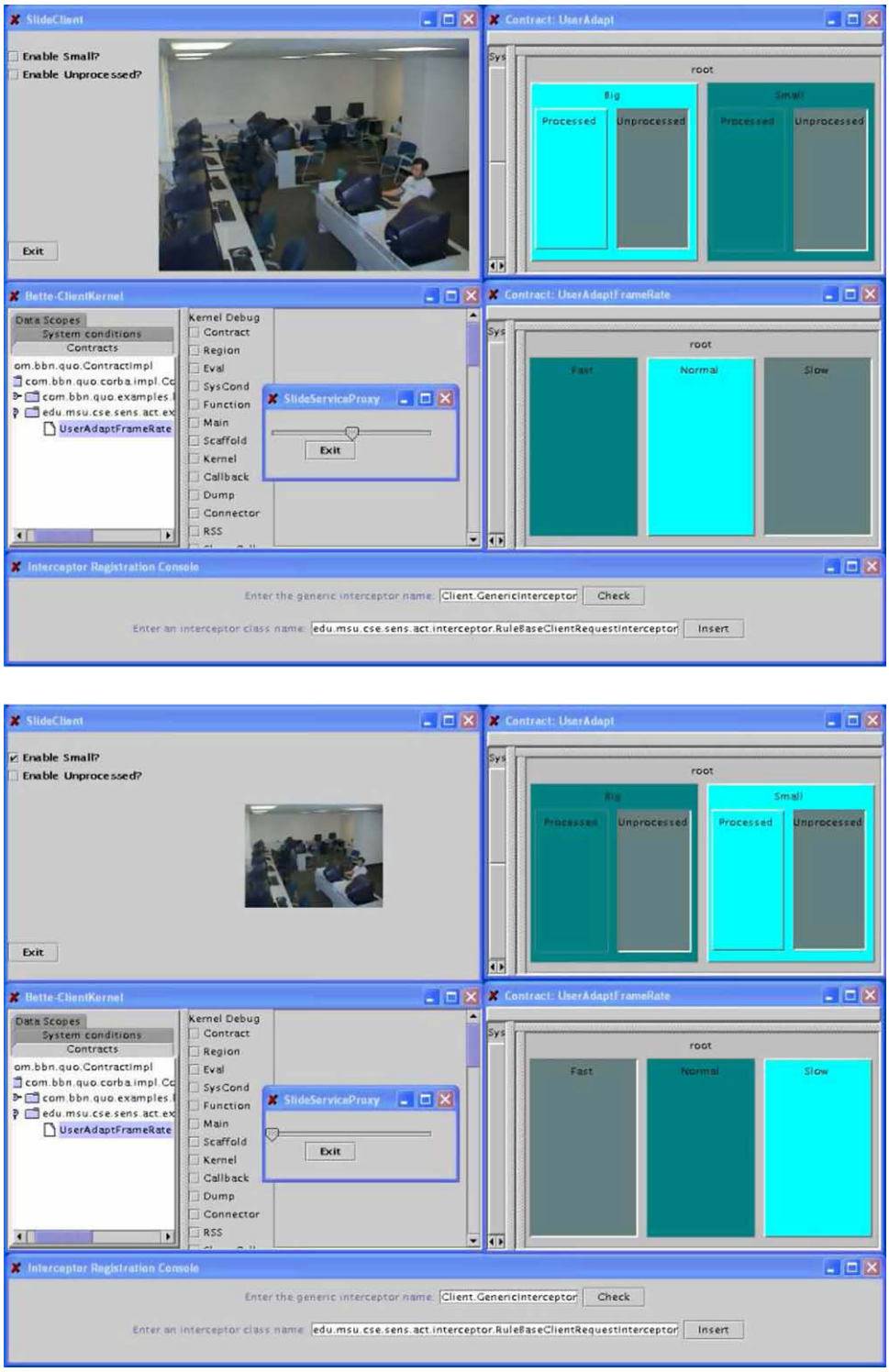


**Fig. 15.** Screen captures of ACT/QuO image-retrieval application: (top) 252 KB version of image displayed; (bottom) 19 KB version of image displayed.

intercept requests and replies for possible modifications [18]. In Section 3.5, we showed how ACT can interact with QuO transparently, enabling new qoskets to be dynamically woven into the application at run time. In a related project, Jacobsen et al. [24] developed an annotated version of the CORBA IDL that enables weaving of semantic properties (such as synchronization and security) into the CORBA skeleton at compile time.

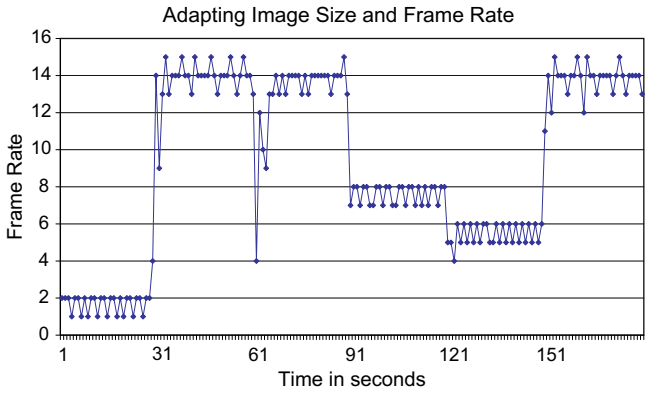


**Fig. 16.** Dynamic adaptation in a ACT/QuO hybrid application.

AspectIX [25,26] is an aspect-oriented distribution middleware that is based on the distributed object model [27], in which an object comprises multiple fragments distributed across nodes. AspectIX enables dynamic weaving of non-functional aspects into object fragments. Although AspectIX is CORBA compliant, its dynamic adaptation feature cannot be used when it interoperates with other non-AspectIX, but CORBA-compliant ORBs. To solve this problem, ACT could be used as a framework gateway that hosts fragments of a distributed object at the non-AspectIX ORBs. Squirrel [28] is an adaptive distribution middleware, specialized for streaming data, that supports QoS for multimedia applications. Again, ACT could be used as a gateway that enables interoperation among non-Squirrel and Squirrel ORBs. Specifically, ACT can enable non-Squirrel ORBs to accept and use *smart proxies* [29] transparently so that they could better communicate with Squirrel ORBs.

### 4.2. Reflective middleware

Reflective middleware uses computational reflection to enable inspection and modification of middleware dynamically during application execution [30]. DynamicTAO [31] and UIC [32] are CORBA-compliant reflective ORBs that employ the component-configurator pattern [33] to support dynamic adaptation. OpenORB [34,35] is a reflective ORB that provides explicit binding of remote objects and enables unanticipated dynamic adaptation using structural and behavioral reflection [36]. The Coyote project [37] also addresses unanticipated dynamic adaptation in distributed applications using Iguana/J. ZEN [38] is a Java ORB that uses Java reflection and the virtual component pattern [39] to provide a minimal-footprint ORB that loads ORB components on demand. To exploit the adaptive features provided by these ORBs, one must use the same ORB in all the autonomous programs that constitute the CORBA application. ACT could be used as a gateway between a non-reflective CORBA-compliant ORB and a reflective ORB, as well as between two reflective ORBs of different types, to enable interoperation while exploiting the adaptive features of the reflective ORBs. To do so, ACT could host different reflective ORBs transparently while intercepting all CORBA requests, replies, and exceptions and passing them to the appropriate reflective ORB.

### 4.3. Intercepting middleware

The concept of transparently intercepting CORBA requests and replies has been used in several projects. Friedman et al. [40,41] use CORBA portable interceptors [12] to enhance the client side of a CORBA application by introducing proxies that can cache replies and forward requests to other CORBA objects. This work is among the first to exploit CORBA portable interceptors for transparent adaptation. In the IRL project, Baldoni et al. [42,43] use portable interceptors to transparently introduce their implementation of fault-tolerant CORBA [44,12] to CORBA-compliant ORBs. Moser et al. [45,46] also use an interception-based approach to transparently introduce their implementation of fault-tolerant CORBA (Eternal [46] over Totem [47]) to CORBA applications. Eternal, however, employs an operating-system interception-based approach instead of using CORBA portable interceptors. In the ALICE project, Haahr et al. [48] use *mobility gateways*, which are proxies at the edge of wired network, to support mobility of CORBA applications by intercepting requests to/from mobile hosts. In general, the above projects focus on modifying program behavior in a particular way, for example, to enhance fault-tolerance. In contrast, ACT uses the concept of generic interceptors to enable adaptation of different types (security, fault-tolerance, QoS, and mobility) in ways that were not necessarily anticipated at application development time. Moreover, generic interception enables ACT to be used as a framework gateway to facilitate interoperation of multiple solutions.

We note that despite the name similarity of our ACT framework and the IBM ACT, there is no relation between these two technologies. The IBM ACT stands for Advanced Connectivity Technology and is an alternative to current keyboard, video, and mouse (KVM) solutions for monitoring multiple racks with multiple servers from one or more consoles; effectively, eliminating the need for long and bulky KVM cables.

## 5. Conclusions

In this paper, we described ACT, an extension of *Transparent Shaping* [49] in CORBA. ACT can be used to produce

families of adaptable program from existing CORBA programs. Specifically, ACT can be used to develop new adaptive CORBA frameworks and to enhance existing frameworks with adaptive functionality and interoperability features. ACT can adapt legacy CORBA applications at run time without the need to modify or recompile their source code. We developed ACT/J, an instance of ACT in Java. A case study was conducted, where we used ACT/J to introduce three types of autonomic behavior to an existing image-retrieval application used for surveillance. The results of our experiments show that the overhead introduced by ACT is negligible. We also showed that ACT can enable transparent integration of new adaptive code into extant QuO applications.

## Further information

This work is part of the ONR-supported RAPIDware project. A number of related papers and software downloads, including the ACT/J prototype, are available at the project website: http://www.cse.msu.edu/rapidware. Further information on the Software Engineering and Network Systems Laboratory can be found at the following URL: http://www.cse.msu.edu/sens. For more information about Transparent Shaping refer to the following URL: http://www.cis.fiu.edu/acrl.

## Acknowledgements

This work was supported in part by the US Department of the Navy, Office of Naval Research (Grant No. N00014-01-1-0744), by National Science Foundation (Grants CCR-9912407, EIA-0000433, EIA-0130724, 1038, ITR-0313142, CCR-9901017, OISE-0730065, HRD-0833093, and OCI-0636031), by U.S. Army (Grant W911WF-08-1-0495), by a Quality Fund Grant from Michigan State University, and by IBM.

## References

[1] Application Integration & Web Services Summit 2004. Gartner, May 2004.
[2] J. Wilber, Q&A with industry analysts: How are e-business trends impacting developers and development teams? The Rational Edge, February 2004.
[3] The Common Object Request Broker: Architecture and Specification Revision 2.0, Object Management Group, Framingham, Massachusett, July 1995.
[4] J. Siegel, OMG overview: CORBA and the OMA in enterprise computing, Communications of the ACM 41 (10) (1998) 37–43.
[5] Object Management Group, CORBA success stories, available at URL: <http://www.corba.org/success.htm>.
[6] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50.
[7] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, B.H.C. Cheng, Composing adaptive software, IEEE Computer (2004) 56–64.
[8] K. van den Berg, J.M. Conejero, J. Hernàndez, Analysis of crosscutting in early software development phases based on traceability, in: Transactions on Aspect-Oriented Software Development (TAOSD), Springer-Verlag, Berlin, Heidelberg, 2007, pp. 73–104.
[9] J.M. Conejero, K. van den Berg, J. Hernàndez, Disentangling crosscutting in aosd: Formalization based on a crosscutting pattern, Jornadas de Ingenieria del Software y Bases de Datos, 2006, pp. 325–334, ISBN 84-95999-99-4.
[10] S.M. Sadjadi, P.K. McKinley, 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.
[11] S.M. Sadjadi, P. McKinley, Transparent self-optimization in existing CORBA applications, in: Proceedings of the International Conference on Autonomic Computing (ICAC-04), New York, NY, May 2004, pp. 88–95.
[12] The Common Object Request Broker: Architecture and Specification Version 3.0, Object Management Group, Framingham, Massachusett, July 2003.
[13] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri, Generic support for distributed applications, IEEE Computer 33 (3) (2000) 68–76.
[14] ORBacus for C++ and Java version 4.1.0, IONA Technologies Inc., 2001.
[15] G. Brose, N. Noffke, JacORB 1.4 documentation, Freie UniversitSt Berlin and Xtradyne Technologies AG, Tech. Rep., August 2002.
[16] D.C. Schmidt, D.L. Levine, S. Mungee, The design of the TAO real-time object request broker, Computer Communications 21 (4) (1998) 294–324.
[17] J. Zinky, J. Loyall, R. Shapiro, 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.
[18] N. Wang, C. Gill, D. Schmidt, A. Gokhale, B. Natarajan, J. Loyall, R. Schantz, C. Rodrigues, Qos-enabled middleware, in: Qusay H. Mahmoud (Ed.), Chapter in Middleware for Communications, vol. 3 (1), 2004.
[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS, vol. 1241, Springer-Verlag, 1997.
[20] R. Schantz, J. Loyall, M. Atighetchi, P. Pal, Packaging quality of service control behaviors for reuse, in: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, April 2002.
[21] R. Pawlak, R. Douence (Eds.), The First Workshop on Aspect-Oriented Middleware Development (AOMD 2005), November 2005.
[22] O. de Moor, G. Kiczales (Eds.), Sixth International Conference on Aspect-Oriented Software Development (AOSD-2007), March 2007.
[23] J.A. Zinky, D.E. Bakken, R.E. Schantz, Architectural support for quality of service for CORBA objects, Theory and Practice of Object Systems 3 (1) (1997).
[24] H.-A. Jacobsen, B. Kraemer, A design pattern based approach for generating synchronization adaptors from annotated IDL, in: Proceedings of the IEEE International Conference on Automated Software Engineering, 1998, pp. 63–72.
[25] F.J. Hauck, U. Becker, E.M.M. Geier, U. Rastofer, M. Steckermeier, Aspectix: a quality-aware, object-based middleware architecture, in: Proceedings of the 3rd IFIP International Conference on Distrib. Appl. and Interoperable Sys. – DAIS, 2001.
[26] R. Kapitza, F. Hauckn, H. Reiser, Decentralized, adaptive services: the aspectix approach for a flexible and secure grid environment, in: Proceedings of the GSEM 2004 Conferences, GSEM, Erfurt, Germany, November 2004.
[27] M. van Steen, P. Homburg, A.S. Tanenbaum, The architectural design of Globe: a wide-area distributed system, Vrije Universiteit, Amsterdam, The Netherlands, Tech. Rep. 422, March 1997.
[28] R. Koster, A.P. Black, J. Huang, J. Walpole, C. Pu, Thread transparency in information flow middleware, Software-Practise and Experience 33 (4) (2003) 321–349.
[29] N. Wang, K. Parameswaran, D.C. Schmidt, O. Othman, Evaluating meta-programming mechanisms for ORB middleware, IEEE Communications Magazine, Special Issue on Evolving Communications Software: Techniques and Technologies, October 2001.
[30] F. Kon, F. Costa, G. Blair, R.H. Campbell, The case for reflective middleware, Communications of the ACM 45 (6) (2002) 33–38.
[31] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, R.H. Campbell, Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB, in: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000), New York, April 2000.
[32] M. Roman, F. Kon, R.H. Campbell, Reflective middleware: from your desk to your hand, IEEE Distributed Systems Online 2 (5) (2001).
[33] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, vol. 2, John Wiley, 2001.
[34] C.-F. Sørensen, M. Wu, T. Sivaharan, G.S. Blair, P. Okanda, A. Friday, H.A. Duran-Limon, A context-aware middleware for applications in

mobile ad hoc environments, in: Middleware for Pervasive and Ad hoc Computing, 2004, pp. 107–110.
[35] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, An architecture for next generation middleware, in: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, September 1998.
[36] G. Blair, G. Coulson, N. Davies, Adaptive middleware for mobile multimedia applications, in: Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video, 1997, pp. 259–273.
[37] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: Proceedings of the 16th European Conference on Object-Oriented Programming, June 2002.
[38] R. Klefstad, D.C. Schmidt, C. O'Ryan, Towards highly configurable real-time object request brokers, in: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, April–May 2002.
[39] A. Corsaro, D. Schmidt, R. Klefstad, C. O'Ryan, Virtual component a design pattern for memory constrained embedded applications, in: Proceedings of the Ninth Conference on Pattern Language of Programs (PLoP 2002), 2002.
[40] R. Friedman, E. Hadad, Distributed wisdom: analyzing distributed-system performance–latency vs. throughput, IEEE Distributed Systems Online 7 (1) (2006) 1.
[41] R. Friedman, E. Hadad, Client side enhancements using portable interceptors, in: Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-time Dependable Systems, January 2001.
[42] C. Marchetti, R. Baldoni, S. Tucci-Piergiovanni, A. Virgillito, Fully distributed three-tier active software replication, IEEE Transactions on Parallel and Distributed Systems 17 (7) (2006) 633–645.
[43] R. Baldoni, C. Marchetti, A. Termini, Active software replication through a three-tier approach, in: Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems, Osaka, Japan, October 2002, pp. 109–118.
[44] P. Narasimhan, T. Dumitras, A.M. Paulos, S.M. Pertet, C.F. Reverte, J.G. Slember, D. Srivastava, Mead: support for real-time fault-tolerant CORBA, Concurrency – Practice and Experience 17 (12) (2005) 1527–1545.
[45] P. Narasimhan, L.E. Moser, P.M. Melliar-Smith, Eternal – a component-based framework for transparent fault-tolerant CORBA, Software Practice and Experience 32 (2002) 771–788.
[46] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, V. Kalogeraki, The eternal system: an architecture for enterprise applications, in: Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99), July 1999.
[47] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, C.A. Lingley-Papadopoulis, T.P. Archambault, The Totem system, in: Proceedings of the 25th International Symposium on Fault Tolerant Computing, Pasadena, California, 1995, pp. 61–66.
[48] M. Haahr, R. Cunningham, V. Cahill, Supporting CORBA applications in a mobile environment, in: Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking, 1999.
[49] S.M. Sadjadi, P.K. McKinley, B.H. Cheng, Transparent shaping of existing software to support pervasive and autonomic computing, 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.

**Dr. Masoud Sadjadi** received a B.S. degree in Hardware Engineering in 1995, a M.S. degree in Software Engineering in 1999, and a Ph.D. degree in Computer Science from Michigan State University in 2004. He is currently an assistant professor in the School of Computing and Information Sciences at Florida International University. He is the Co-Director of the Autonomic Computing Research Laboratory (ACRL, http://acrl.cis.fiu.edu), the leader of several projects under the Latin American Grid (LA Grid, http://latinamericangrid.org), an active Co-PI of FIU Partnership for International Research and Education (PIRE, http://pire.fiu.edu/), and one of the Co-Founders of the Global CyberBridges (GCB, http://cyberbridges.net), the Research and Education for Undergraduates (REU, http://www.cis.fiu.edu/reu), and the Communication Virtual Machine (CVM, http://www.cis.fiu.edu/cvm) projects at FIU. He has extensive experience in software development and leading large scale software projects. Currently, he is collaborating with researcher in eight countries and is leading several international research projects. He was the Program Co-Chair of the IEEE ICNSC 2008, has served in organizational and program committees of several international conferences and workshop, and has served as a referee for several IEEE and SP&E journals. His current research interests include Software Engineering, Distributed Systems, and High-Performance Computing with the focus on Autonomic, Pervasive, and Grid Computing. He has published more than 50 papers and is PI or Co-PI of 10 grants from NSF, IBM, and FIU for total of over $4.3 million. He is a member of the IEEE.

**Philip K. McKinley** received the B.S. degree in mathematics and computer science from Iowa State University in 1982, the M.S. degree in computer science from Purdue University in 1983, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1989. Dr. McKinley is currently a Professor in the Department of Computer Science at Michigan State University, where he has been on the faculty since 1990. He was a member of technical staff at Bell Laboratories in Naperville, Illinois from 1982–1990, on leave of absence 1985–1989. Dr. McKinley has served as an Associate Editor for IEEE Transactions on Parallel and Distributed Systems and was co-chair of the program committee for the 2003 IEEE International Conference on Distributed Computing Systems. His current research interests include adaptive middleware, collaborative applications, mobile computing, and group communication protocols. He is a member of the IEEE and ACM.