9

Transparent Autonomization in Composite Systems

S. Masoud Sadjadi and Philip K. McKinley

CONTENTS

9.1 Introduction

The ever-increasing complexity of computing systems has been accompanied by an increase in the complexity of their management. Contributing factors include the increasing size of individual networks and the dramatic growth of the Internet, the increasing heterogeneity of software and hardware components, the deployment of new networking technologies, the need for mobile access to enterprise data, and the emergence of pervasive computing. In this chapter, we focus on the management complexity resulting from integrating existing, heterogeneous systems to support corporate-wide, as well as Internet-wide, connectivity of users, employees, and applications.

Autonomic computing [1] promises to solve the management problem by embedding the management of complex systems inside the systems themselves. Instead of requiring low-level instructions from system administrators in an interactive and tightly coupled fashion, such self-managing systems require only high-level human guidance—defined by goals and policies—to work as expected. However, if the code for self-management and application integration is entangled with the code for the business logic of the original systems, then the complexity of managing the integrated system may actually increase, contradicting the purpose of autonomic computing.

To integrate heterogeneous applications, possibly developed in different programming languages and targeted to run on different platforms, requires conversion of data and commands between the applications. The advent of middleware—which hides differences among programming languages, computing platforms, and network protocols [2–4]—in the 1990s mitigated the difficulty of application integration. Indeed, the maturity of middleware technologies has produced several successful approaches to *corporate-wide* application integration [5,6], where applications developed and managed by the same corporation are able to interoperate with one another.

Ironically, the difficulty of application integration, once alleviated by middleware, has reappeared with the proliferation of *heterogeneous* middleware technologies. As a result, there is a need for a "middleware for middleware" to enable Internet-wide and business-to-business application integration [7]. Successful middleware technologies such as Java Remote Method Invocaton (RMI), Common Object Request Broker Architecture (CORBA), and DCOM/.NET Remoting have been used to integrate corporate-wide applications. However, such middleware technologies are often unable to integrate applications managed by different corporations connected through the Internet. The reasons are twofold: (1) different corporations select different middleware technologies, which are more appropriate to integrate their own applications; and (2) middleware packets often cannot pass through Internet firewalls.

Web services [8] offer one approach to addressing these problems. A *web service* is a program delivered over the Internet that provides a service described in the Web Service Description Language (WSDL) [9] and communicates with other programs using messages in Simple Object Access Protocol (SOAP) [10]. WSDL and SOAP are both independent of specific platforms, programming languages, and middleware technologies. Moreover, SOAP leverages the optional use of the HTTP protocol, which can bypass firewalls, thereby enabling Internet-wide application integration.

Although Web services have been successfully used to integrate heterogeneous applications, by themselves they do not provide a *transparent* solution. A challenging problem is to enable integration of existing applications without entangling the integration and self-management concerns with the business logic of the original applications. In this chapter, we describe a technique to enable self-managing behavior to be added to composite systems transparently, that is, without requiring manual modifications to the existing

code. The technique uses *transparent shaping* [11], developed previously to enable dynamic adaptation in existing programs, to weave self-managing behavior into existing applications. We demonstrate that combining transparent shaping with Web services provides an effective solution to the transparent application integration problem.

The remainder of this chapter is organized as follows. Section 9.2 provides background on Web services. Section 9.3 describes two approaches to transparent application integration through Web services. Section 9.4 overviews two instances of transparent shaping that can be used in application integration. Section 9.5 presents a case study, where we use these transparent shaping techniques to integrate two existing applications, one developed in Microsoft .NET and the other in CORBA, in order to construct a fault-tolerant surveillance application. Section 9.6 discusses related work, and Section 9.7 summarizes the chapter.

9.2 Web Services Background

A service-oriented architecture [8], as depicted in Figure 9.1, is composed of at least one *provider program*, which is capable of performing the actions associated with a service defined in a service description, and at least one *requester program*, which is capable of using the service provided by a service provider.[1] In this model, we assume that a program is executed inside a process, with a boundary distinguishing local and remote interactions, and is composed of a number of software components, which are units of software composition hosted inside a program process.[2] A component implementing a service is called a *provider component*, and a component requesting a service is called a *requester component*.

Figure 9.1 also shows that the application-to-application (A2A) interaction is accomplished through the use of a middleware technology over a network. The network can be the Internet, an intranet, or simply an interprocess communication (IPC) facility. In the case of Web services, the middleware is composed of two layers: a SOAP messaging layer governed by a WSDL layer (described below). *Web services* are software programs accessible over the Internet by other programs using the SOAP messaging protocol and service descriptors defined in WSDL.

SOAP. SOAP [10] is an XML-based messaging protocol designed to be independent of specific platforms, programming languages, middleware

[1] We use the terms "provider program" and "requester program" instead of the terms "provider agent" and "requester agent," as used in [8], to avoid confusion with agents in agent-based systems.

[2] The example programs provided in this chapter are all developed in object-oriented languages. For simplicity, the terms "component" and "object" have been used interchangeably. However, this does not imply that a service-oriented system must either be implemented using object-oriented languages or be designed using an object-oriented paradigm.

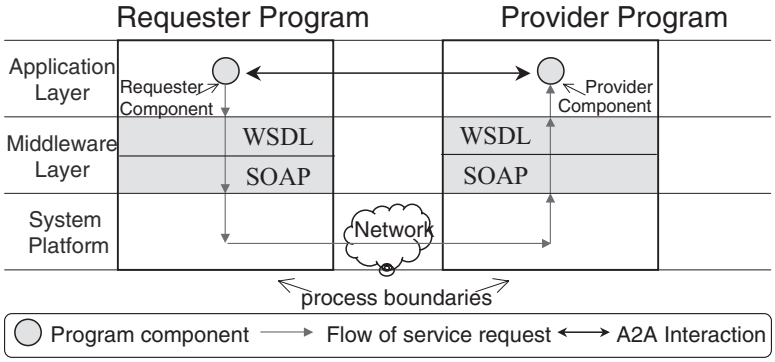

FIGURE 9.1
A simplified Web service architecture.

technologies, and transport protocols. SOAP messages are used for interactions among Web service providers and requesters. Unlike object-oriented middleware such as CORBA, which requires an object-oriented model of interaction, SOAP provides a simple message exchange among interacting parties. As a result, SOAP can be used as a layer of abstraction on top of other middleware technologies (effectively providing a middleware for middleware).

A SOAP message is an XML document with one element, called an envelope, and two children elements, called the header and body. The contents of the header and body elements are arbitrary XML. Figure 9.2 shows the structure of a SOAP message. The header is an optional element, whereas the body is not optional; there must be exactly one body defined in each SOAP message. To provide the developers with the convenience of a procedure-call abstraction, a pair of related SOAP messages can be used to realize a request and its corresponding response. SOAP messaging is *asynchronous*, that is, after sending a request message, the service requester will not be blocked waiting for the response message to arrive. For more information about details of SOAP messages, please refer to [10,12].

```
 1 <?xml version=``1.0" encoding=``UTF-8" ?>
 2 <soap:Envelope xmlns:soap=
 3   ``http://schemas.xmlsoap.org/soap/envelope/ ... >
 4   <soap:Header>
 5     <!-- Header contents in defined in arbitrary XML. -->
 6   </soap:Header>
 7   <soap:Body>
 8     <!-- Body contents in defined in arbitrary XML. -->
 9   </soap:Body>
10 </soap:Envelope>
```

FIGURE 9.2
SOAP message structure.

WSDL. WSDL [9] is an XML-based language for describing valid message exchanges among service requesters and providers. The SOAP messaging protocol provides only basic communication and does not describe what pattern of message exchanges are required to be followed by specific service requesters and providers. WSDL addresses this issue by describing an interface to a Web service and providing the convenience of remote procedure calls (or more complicated protocols). For more information about details of WSDL, please refer to [9,12].

9.3 Transparent Application Integration

Several different approaches have been described in the literature to integrate applications. Regardless of the specific technique, integration of two heterogeneous applications requires translating the syntax and semantics of the two applications, typically during execution. Providing direct translations for N heterogeneous middleware technologies requires $(N^2 - N)/2$ translators to cover all possible pairwise interactions. Using a common language reduces the number of translators to N, assuming that one side of the interaction (either requester or provider program) always uses the common language. Web services provide one such language. Depending on *where* the translation is performed (e.g., inside or outside the requester and provider programs), we distinguish two approaches to transparent application integration: the bridge approach and the transparent shaping approach.

Bridge Approach. An intuitive approach to transparent application integration is to use *bridge programs*, which sit between requesters and providers, intercepting the interactions and translating them from application-specific middleware protocols to Web services protocols, and vice versa. The architecture for this approach is illustrated in Figure 9.3. A bridge program hosts one or more *translator components*, which encapsulate the logic for translation. A translator component plays the role of a provider component for the requester component, as well as the role of a requester component for the provider component.

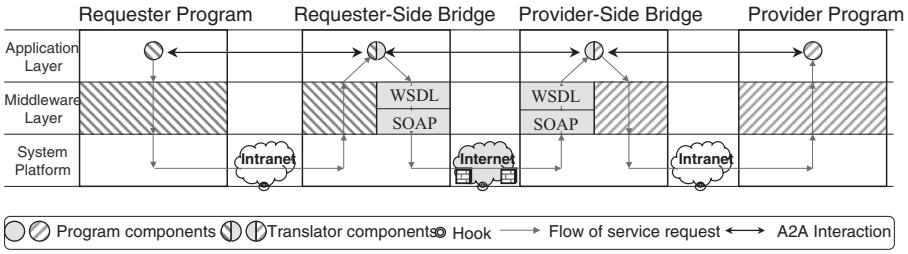

FIGURE 9.3
Transparent application integration using bridge programs.

Using this architecture is beneficial for the following reasons. First, hosting translator components inside a separate process (the bridge program) does not require modifications to the requester and provider programs. Second, a bridge program can host several translator components, where each translator component may provide translation to one or more requester and provider programs. Third, the localization of translator components in one location (the bridge program) simplifies the maintenance of application integration. For example, security policies can be applied to the bridge program once, and will be effective to all the translator components hosted by the bridge.

The main disadvantage of this architecture is the overhead imposed on the interactions due to process-to-process redirection (in case the bridge programs are located on the same machine as the requester and/or provider) or machine-to-machine redirection (in case the bridge programs are located on separate machines). Other disadvantages include a potential single point of failure and a communication/processing bottleneck at the bridge. Furthermore, this approach may not even be possible in some situations—for example, if the provider address is hard-coded in the requester program.

Transparent Shaping Approach. To avoid these problems, the translator components can instead be hosted *inside* the requester and provider programs, as illustrated in Figure 9.4. However, providing intercepting and redirecting interactions transparently to the application is challenging. *Transparent shaping* provides a solution to this problem by generating adaptable programs from existing applications. We call a program *adaptable* if its behavior can be changed with respect to the changes in its environment or its requirements. An adaptable program can be thought of as a managed element, as described in [1].

We developed transparent shaping originally to enable reuse of existing applications in environments whose characteristics were not necessarily anticipated during the design and development [13,14]. An example is porting applications to wireless networks, which often exhibit relatively high packet loss rates and frequent disconnections. In this chapter, however, we show how transparent shaping can be used to enable transparent application integration. The integration is performed in two steps.

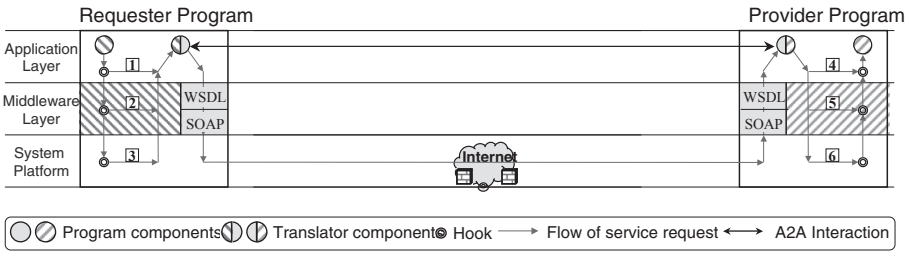

FIGURE 9.4
Transparent application integration using transparent shaping.

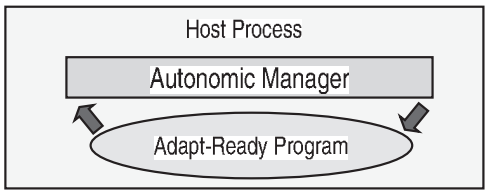

FIGURE 9.5
Transparent adaptation of an adapt-ready program at runtime.

In the first step, an *adapt-ready* program is produced at compile, startup, or load time using static transformation techniques. An adapt-ready program is a managed element whose behavior is initially equivalent to the original program but which can be adapted at runtime by inserting or removing adaptive code at certain points in the execution path of the program, called *sensitive joinpoints*. For application integration, we are interested only in those joinpoints related to remote interactions. To support such operations, the first step of transparent shaping weaves interceptors, referred to as *hooks*, at the remote interaction joinpoints. As illustrated in Figure 9.4, hooks may reside inside the program code itself (arrows 1 and 4), inside its supporting middleware (arrows 2 and 5), or inside the system platform (arrows 3 and 6). Example techniques for implementing hooks include weaving aspects into the application (compile time) [14], inserting portable interceptors into a CORBA program [15] (startup time), and byte-code rewriting in a virtual machine [16] (load time).

In the second step, executed at runtime, the hooks in the adapt-ready program are used by an autonomic manager to redirect the interactions to adaptive code, which in this case implements the translator. As illustrated in Figure 9.5, the autonomic manager may introduce new behavior to the adapt-ready program according to the high-level user policies by inserting or removing adaptive code using the generic hooks.

9.4 Transparent Shaping Mechanisms

In this section, we briefly describe two concrete instances of transparent shaping. The first one is a language-based approach that uses a combination of aspect weaving and meta-object protocols to introduce dynamic adaptation to the application code directly. The second one is a middleware-based approach that uses middleware interceptors as hooks. Both instances adhere to the general model described above.

Language-Based Transparent Shaping. Transparent Reflective Aspect Programming (TRAP) [14] is an instance of transparent shaping that supports dynamic adaptation in existing programs developed in class-based, object-oriented programming languages. TRAP uses generative techniques to create

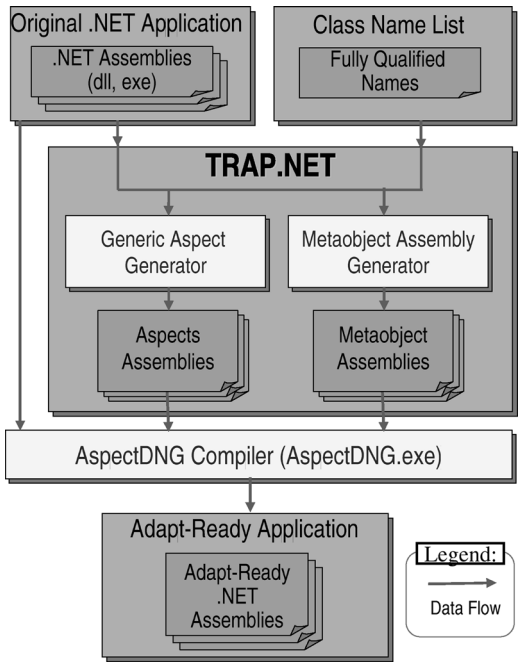

FIGURE 9.6
TRAP.NET compile-time process.

an adapt-ready application, without requiring any direct modifications to the existing programs. To validate the TRAP model, previously we developed TRAP/J [14], which supports dynamic adaptation in existing Java programs. To support existing programs developed in C++, members of our group have implemented TRAP/C++ using compile-time meta-object protocols supported by Open C++ [17].

For the case study described in Section 9.5, we developed TRAP.NET, which supports dynamic adaptation in .NET applications developed in any .NET programming language (e.g., C#, J#, VB, and VC++). As illustrated in Figure 9.6, the developer selects at compile time a subset of classes in the existing .NET assemblies[3] that are to be reflective at runtime. A class is *reflective* at run time if its behavior (i.e., the implementation of its methods) can be inspected and modified dynamically [18]. Since .NET does not support such functionality inherently, TRAP.NET uses generative techniques to produce adapt-ready assemblies with hooks that provide the reflective facilities for the selected classes. Next, we use *AspectDNG* version 0.6.2, a recently released .NET aspect weaver [19], to weave generated aspect and meta-object assemblies into the original application. Finally, we execute the adapt-ready

[3] A .NET assembly is simply a .NET executable file (i.e., a .EXE file) or a .NET library file (i.e.,a .DLL file).

application together with an autonomic manager using a host application (explained in Section 9.5.3).

As the adapt-ready assemblies execute, the autonomic manager may introduce new behavior to the adapt-ready assemblies according to the high-level user policies by insertion and removal of adaptive code via interfaces to the reflective classes. Basically, the hooks inside a reflective class wrap the original methods, intercept all incoming calls, and can forward the calls to new implementations of the methods, as needed. The new implementations can be inserted dynamically by the autonomic manager.

Middleware-Based Transparent Shaping. For those applications written atop a middleware platform, transparent shaping can be implemented in middleware instead of this application itself. The *Adaptive CORBA Template (ACT)* [13,20] is an instance of transparent shaping that enables dynamic adaptation in existing CORBA programs. ACT enhances CORBA Object Request Brokers (ORBs) to support dynamic reconfiguration of middleware services transparently not only to the application code, but also to the middleware code itself. Although ACT itself is specific to CORBA, the concepts can be applied to many other middleware platforms. To evaluate the performance and functionality of ACT, we constructed a prototype of ACT in Java, called *ACT/J* [13,20]. To support CORBA programs developed using C++ ORBs, we plan to develop ACT/C++. In addition, we are planning to develop similar frameworks for Java/RMI and Microsoft's .NET.

Figure 9.7 shows the flow of a request/reply sequence in a simple CORBA application using ACT/J. For clarity, details such as stubs and skeletons are not shown. ACT comprises two main components: a 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

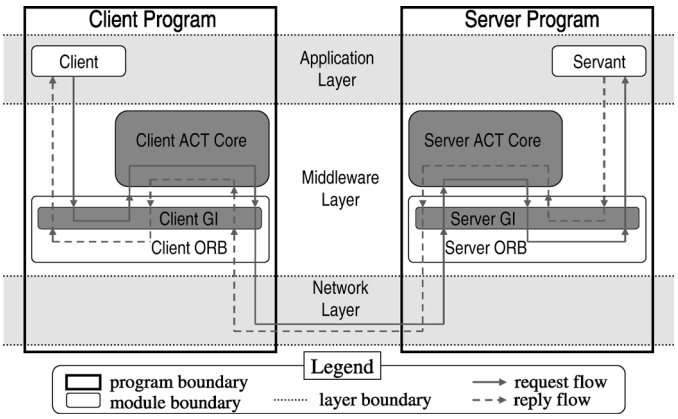

FIGURE 9.7
ACT configuration in the context of a simple CORBA application.

interceptor intercepts all the incoming requests and outgoing replies (or exceptions) and forwards them to its ACT core. The CORBA application is *adapt-ready* if a generic interceptor is registered with all its ORBs at startup time.

Implementing transparent shaping in middleware, as in ACT, can produce greater transparency than a language-based approach such as TRAP, which requires recompilation of the application. To produce an adapt-ready version of an existing CORBA program, we do not need to transform the original program as we do in TRAP. By introducing generic hooks inside the ORB of a CORBA application at startup time, we can intercept all CORBA remote interactions. Specifically in ACT/J, a generic host program (this program is part of the ACT/J framework) is first instructed to follow the settings in a configuration file.[4] The configuration file instructs the host program to incorporate generic CORBA portable interceptors [15] into the ORB of the host program. Next, the host program loads both the original application together with an autonomic manager.

Later at runtime, these hooks can be used by an autonomic manager to insert and remove adaptive code with respect to the adapt-ready program, which in turn can adapt the requests, replies, and exceptions passing through the ORBs. In this manner, ACT enables runtime improvements to the program in response to unanticipated changes in its execution environment.

9.5 Case Study: Fault-Tolerant Surveillance

To demonstrate how transparent shaping and Web services can be used to integrate existing applications, while introducing new autonomic behavior, we conducted a case study. Specifically, we created a fault-tolerant surveillance application by integrating existing .NET and CORBA image retrieval applications and adding a self-management component to switch among the two image sources in response to failures. The integration and self-management code is transparent to the original applications. The resultant integrated system is a *fault-tolerant* surveillance application. In the remainder of this section, we briefly introduce each of the two applications, review our strategy for integration and self-management, and describe the details of the integration process and the operation of the integrated system.

9.5.1 Existing Applications

The Sample Grabber Application. The first application is a .NET sample grabber application (called `SampleGrabberNET`). Basically, it captures a video stream from a video source (e.g., a WebCam) and displays it on the screen. In addition, it allows the user to grab a live frame and save it to a bitmap file. This application is a .NET standalone application written in

[4] The corresponding command line instruction used at startup time is the following: java Host.class Host.class.config OriginalApplication.class AutonomicManger.class

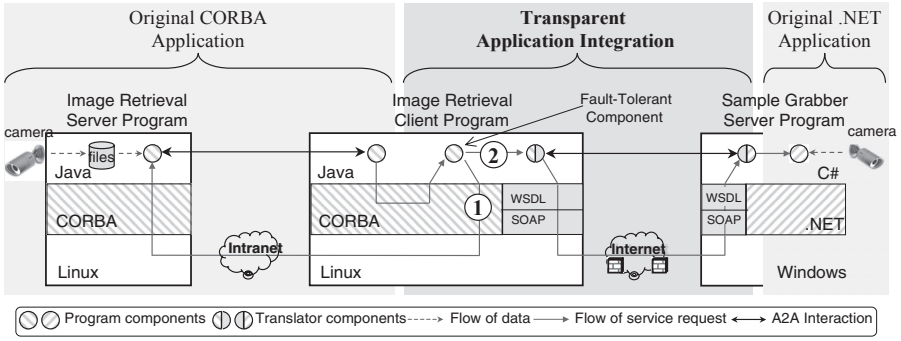

FIGURE 9.8
The configuration of a fault-tolerant surveillance application.

C# and freely available at the Code Project web site (`http://www.codeproject.com`). It is part of the `DirectShow.NET` framework developed by NETMaster.[5]

The Image Retrieval Application. The second application is a CORBA image retrieval application written in Java. It has two parts, a *client program* (called `SlideClient`) that requests and displays images, and a *server program* (called `SlideService`) that stores images and replies to the client program requests. The client program continuously sends requests to the server program asking for images and displays each returned image. In our implementation, the images maintained by the server program are live shots taken periodically using a WebCam and stored in a file. This application was developed by BBN Technologies and is distributed with the QuO framework [21]. The image retrieval application by itself can benefit from the QuO framework, which supports several types of adaptive behavior. In this study we disabled the QuO framework and used the application only as a CORBA application.

9.5.2 Integration and Self-Management Strategy

Our goal is to construct a surveillance application that can use either application as the source of images. For example, the application should enable the CORBA client application to retrieve live images from the .NET frame grabber application, when the CORBA server application is not available (e.g., due to server crash, network disconnection, etc.). Figure 9.8 shows the configuration of the integrated application. The .NET frame grabber application plays the role of a provider program and must be exposed as a frame grabber Web service. On the other side, the client program of the image retrieval

[5] NETMaster is the name for a group of developers who are actively contributing to the Code Project. NETMaster uses the interfaces provided in the `DirectShow.NET` framework to interoperate with DirectShow. The Code Project (`http://www.codeproject.com`) is a repository for a large number of free C++, C# and .NET articles, code snippets, discussions, and news on the Internet.

```
 1 // The host application defined in  Host.cs
 2 public class Host  {
 3   static private string configFilename, managedElement, autonomicManager;
 4   public static void Main(string [] args)  {
 5   if (args.Length != 3) return;
 6   configFilename = args[0]; managedElement = args[1]; autonomicManager = args[2];
 7   try  {
 8     RemotingConfiguration.Configure(configFilename);
 9     AppDomain ad = AppDomain.CurrentDomain;
10     ad.ExecuteAssembly(autonomicManager);
11     ad.ExecuteAssembly(managedElement);
12    } catch(Exception e)  { }
13   String keyState = ``"; keyState = Console.ReadLine();
14  }
15
16 // The configuration file defined in Host.exe.config
17 <configuration> <system.runtime.remoting> <application name="Server">
18   <service>
19     <wellknown mode="Singleton" type="SampleGrabberWebService.
20       SampleGrabberObject, SampleGrabberObject" objectUri="SampleGrabberObject" />
21   </service>
22   <channels> <channel port="9000" ref="http" /> </channels>
23 </application> </system.runtime.remoting> </configuration>
```

FIGURE 9.9
Excerpted code for the Host program hosting the adapt-ready application and its autonomic manager.

application plays the role of a requester program and must be shaped to use both the frame grabber Web service as well as the CORBA server program.

In the rest of this section, we describe how transparent shaping is used to expose the frame grabber application as a Web service, followed by how the image retrieval client program is shaped to use this Web service.

9.5.3 Exposing the Frame Grabber Application

.NET Remoting is an enabler for distributed applications to communicate with one another, thus allowing interprocess communication. We note that the server side of a distributed application that uses .NET remoting can be exposed as a Web service without modification if none of the types to be exposed by the .NET server application is a .NET specific type [22,23]. However, our frame grabber application is a .NET *standalone* application (as opposed to a .NET remoting application). Therefore, no .NET remoting service is exposed by the frame grabber application itself. Hence, we first need to shape the .NET frame grabber application to become a .NET remoting application, and then use it as a Web service.

Following the first step of TRAP.NET (see Figure 9.6), we generate an adapt-ready version of the frame grabber application. To transform this application into a .NET remoting server, we need to weave hooks inside the main class of the application (`MainForm`). Therefore, we list only the name of the main class to be passed to TRAP.NET. The code for .NET remoting and the self-management functionality are located in a separate assembly (`AutonomicManager.exe`). At startup time these two programs are loaded inside another program (`Host.exe`), which is listed in Figure 9.9 (lines 1

to 14).[6] The modified .NET frame grabber program is inside the `SampleGrabberNET.exe` assembly, the .NET remoting code is inside the `AutonomicManager.exe` assembly, and the configuration is inside the `Host.exe.config` file. The excerpted code for the `Host.exe.config` configuration file is listed in Figure 9.9 (lines 16 to 23).

As shown in Figure 9.9, first, the configuration file is parsed and the instructions are followed (line 8), which provides flexibility to configure the `Host` program at startup time (e.g., the port address at which the Web service can be reached can be defined in this configuration file). Next, the `autonomicManager` and the `managedElement` are executed using the .NET reflection facilities (lines 10 and 11).

Now that the provider program is ready to run, we need to generate the Web service description of our provider program (to be used in the shaping of the CORBA client program). We used the .NET framework `SOAPsuds.exe` utility with the `-sdl` option, which generates a WSDL schema file. The excerpted WSDL description is listed in Figure 9.10.

The abstract description part (lines 3 to 16) describes the interface to the Web service using `message` elements (lines 3 to 8), which define what type of messages can be sent to and received from the Web service, and a `portType` element (lines 9 to 16), which defines all the operations that are supported by the Web service. The `GrabFrame` operation (lines 10 to 15) defines the valid message exchange pattern supported by the Web service.

The concrete part of the WSDL description (lines 18 to 33) complements the abstract part using a `binding` element (lines 18 to 26), which describes *how* a given interaction is performed over *what* specific transport protocol, and a `service` element (lines 28 to 33), which describes *where* to access the service. The *how* part describes how marshalling and unmarshalling is performed using the `operation` element inside the `binding` element (lines 21 to 25). The *what* part is described in line 20 using the `transport` attribute. The *where* part is described using the `port` element (lines 29 to 32).

9.5.4 Shaping the Image Retrieval Client

We follow the architecture illustrated in Figure 9.8 to shape the CORBA client program to interoperate with the .NET frame grabber program. We use the ACT/J framework to host a Web service translator in the adapt-ready client application.

Figure 9.11 lists the Interactive Data Language (IDL) description used in the original CORBA image retrieval application. The `SlideShow` interface defines six methods (lines 4 to 9). As listed in Figure 9.12 (lines 21 to 26), all the `read*()` methods defined in the IDL file are mapped to the `GrabFrame()` method of the Web service exposed by the provider program. The `getNumberOfGifs()` method simply returns -1 (line 27) to indicate that

[6] The corresponding command line instruction used at startup time is the following: `Host.exe Host.exe.config AdaptReadyApp lication.exe AutonomicManger.exe`

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <definitions name='SampleGrabberObject' ... > <types> ... </types>
3   <message name='SampleGrabberObject.GrabFrameInput'>
4     <part name='nQuality' type='xsd:int'/>
5   </message>
6   <message name='SampleGrabberObject.GrabFrameOutput'>
7     <part name='return' type='ns2:ArrayOfShort'/>
8   </message>
9   <portType name='SampleGrabberObjectPortType'>
10    <operation name='GrabFrame' parameterOrder='nQuality'>
11      <input name='GrabFrameRequest'
12        message='tns:SampleGrabberObject.GrabFrameInput'/>
13      <output name='GrabFrameResponse'
14        message='tns:SampleGrabberObject.GrabFrameOutput'/>
15    </operation>
16  </portType>
17
18  <binding name='SampleGrabberObjectBinding'
19    type='tns:SampleGrabberObjectPortType'>
20    <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/> ...
21    <operation name='GrabFrame'>
22      <soap:operation soapAction='...'/> ...
23      <input name='GrabFrameRequest'> <soap:body ... /> </input>
24      <output name='GrabFrameResponse'> <soap:body ... /> </output>
25    </operation>
26  </binding>
27
28  <service name='SampleGrabberObjectService'>
29    <port name='SampleGrabberObjectPort' binding='tns:SampleGrabberObjectBinding'>
30      <soap:address location=
31        'http://haydn.cse.msu.edu:9000/Server/SampleGrabberObject'/>
32    </port>
33  </service>
34 </definitions>
```

FIGURE 9.10
The excerpted WSDL description of the sample grabber service.

```
 1 // The slide show interface defined in  SlideShow.idl
 2 module com  { module bbn  { module quo  { module examples  { module bette  {
 3   interface SlideShow  {
 4     void readSmall ( in long gifNumber, out string size, out octetArray buf );
 5    void readSmallProcessed ( in long gifNumber, out string size, out octetArray buf );
 6     void readBig ( in long gifNumber, out string size, out octetArray buf );
 7    void readBigProcessed ( in long gifNumber, out string size, out octetArray buf );
 8     void read ( in long gifNumber, out string size, out octetArray buf );
 9     long getNumberOfGifs ( );
10    };
11  };  };  };  };  };
```

FIGURE 9.11
The slide show IDL file used in the original CORBA application.

the images being retrieved are live images (as opposed to being retrieved from a number of stored images at the server side).

Using the ACT/J framework, the calls to the original CORBA server application are intercepted and redirected to the translator component. The translator component is defined as `SlideService_ClientLocalProxy` class listed in Figure 9.12. First, a reference to the `SampleGrabberObject` Web service is obtained (lines 4 to 13). We used the Java WSDP framework to generate the stub class corresponding to the Web service using the WSDL file, generated in the previous part and listed in Figure 9.10. Next, all calls to the original CORBA object are forwarded to the Web service (lines 14 to 27).

9.5.5 Self-Managed Operation

The configuration of the resulting fault-tolerance surveillance application is illustrated in Figure 9.8. The target setting is to have two (or more) cameras monitoring the same general area (a parking lot, building entrance, etc.). The cameras and supporting software might have been purchased for other purposes and now are being combined to create a fault-tolerant, heterogenous application. Although this application is relatively simple, it demonstrates that it is possible to integrate application in a transparent manner without using bridging.

In our experiments, we first start the adapt-ready .NET frame-grabber application (shown in the right side of Figure 9.8), next the original CORBA slide-server (shown in the left side), and finally the adapt-ready CORBA slide-client application (shown in the middle). We have also implemented a simple user interface that enables a user to enter policies to be followed by the application. By default, the initial policy regarding the data source is to retrieve images from the CORBA server, since our experiments show it is more responsive than the .NET application. If the CORBA server is not available or does not respond for a certain interval (1 second by default), then the client should try to retrieve images from the .NET server. While the images are being retrieved from the .NET server, the client continues to probe the CORBA server to see if it is available. If the CORBA server returns, the client should stop retrieving images from the .NET server and switch to the CORBA

```
1 // The proxy defined in SlideService\_ClientLocalProxy.java
2 public class SlideService\_ClientLocalProxy extends SlideShowPOA
3   implements Serializable, SlideShowOperations  {
4   private SampleGrabberObjectPortType sampleGrabberObject = null;
5   public SlideService\_ClientLocalProxy(ORB orb)  { ...
6     string endpoint = "http://haydn.cse.msu.edu:9000/Server/SampleGrabberObject";
7     try {
8       Stub stub = (Stub)(new SampleGrabberObjectService_Impl().
9         getSampleGrabberObjectPort());
10       stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
11       sampleGrabberObject = (SampleGrabberObjectPortType)stub;
12     } catch (Exception ex) {...}
13   }
14   private byte[] grabFrame( int nQuality )  {
15     byte [] frameByteArray  = null; short[] frameShortArray = null;
16     try { frameShortArray = sampleGrabberObject.GrabFrame( nQuality ); }
17     catch(Exception e) {...}
18     frameByteArray = convertShortArray2ByteArray( frameShortArray );
19     return frameByteArray;
20   } ...
21   public void readBigProcessed(int gifNumber, StringHolder sizeHolder, octetArrayHolder pixelsHolder) {
22     pixHolder.value = grabFrame( 75 ); sizeHolder.value = "big";
23   } ...
24   public void readSmallProcessed(int gifNumber, StringHolder sizeHolder, octetArrayHolder
       pixelsHolder)  {
25     pixelsHolder.value = grabFrame( 50 ); sizeHolder.value = "small";
26   }
27   public int getNumberOfGifs()  { return -1;  }
28 }
```

FIGURE 9.12

Excerpted code for the Web service translator component defined as a proxy object in the ACT/J framework.

server. All these operations are carried out completely transparently to the original CORBA and .NET applications.

Some limited functionality has been provided to enable the user to change the high-level policy via the Graphical User Interface (GUI). For example, the user can change the initial policy and configure the system so that every other image is retrieved from the .NET server. In addition, the user can change the timeout parameter to be other than 1 second. The user can also set the frequency of images being retrieved and ask the system to maintain this frequency. For example, if the user asks for 2 frames per second, then the system automatically monitors the round-trip delay of retrieving images and correspondingly inserts interval delays (if the current frequency is too high) or requests smaller images with lower resolution (if the current frequency is too low). This feature has been tested over a wireless ad hoc network, where a user with the client application is walking about a wireless cell and experiencing different packet loss rates in the wireless network. Although the round-trip time was constantly changing in this experiment, the system was able to maintain the frequency of images being retrieved and displayed.

Finally, we note that once the adapt-ready applications are executed, we can dynamically modify even the self-management functionality itself–for example, introducing a more sophisticated GUI, without the need to stop, modify, compile, and restart the system.

9.6 Related Work

In this section, we review several research projects, standard specifications, and commercial products that support application integration. Based on the transparency and flexibility of the adaptation mechanisms used to support application integration, we identify three categories of related work.

In the first category, we consider approaches that provide transparency with respect to either an existing provider program or an existing requester program, but not both. Therefore, the programs hosting translator components are required to be either developed from scratch or modified directly by a developer. Examples of research projects in this category include the Automated Interface Code Generator (AIAG) [24], the Cal-Aggie Wrap-O-Matic project (CAWOM) [25], and the World Wide Web Factory (W4F) [26]. AIAG [24] supports application integration by providing an interface wrapper model, which enables developers to treat distributed objects as local objects. AIAG is an automatic wrapper generator built on top of JavaSpaces that can be used to generate the required glue code to be used in client programs. CAWOM [25] provides a tool that generates wrappers, enabling command-line systems to be accessed by client programs developed in CORBA. This approach provides transparency for existing command-line systems. Examples of the use of CAWOM include wrapping the JDB debugger, which enables distributed

debugging, and wrapping the Appache Web server, which enables remote administration. Finally, W4F [26] is a Java toolkit that generates wrapper for Web resources. This toolkit provides a mapping mechanism for Java and XML.

In the second category, we consider approaches that provide transparency with respect to both the provider and requester programs using a bridge program hosting the translator components. Although such approaches provide transparency, they suffer from extra overhead imposed by another level of process-to-process or machine-to-machine redirection, as discussed in Section 9.3. An example of a research project in this category is on-the-fly wrapping of Web services [27]. In this project, Web services are wrapped to be used by Java programs developed in Jini [28], a service-based framework originally developed to support integration of devices as services. The wrapping process is facilitated by the `WSDL2Java` and `WSDL2Jini` generator tools, which generate the glue code part of the bridge program and the translator component. A developer is required to complete the code for the bridge and to ensure that the semantics of translations are correct. Using the Jini lookup service, the bridge publishes the wrapped Web service as a Jini service, which can be used transparently by Jini client programs.

We consider transparent shaping in a third category. Similar to the approaches in the second category, transparent shaping provides transparency to both provider and requester programs, and in addition provides flexibility with respect to *where* the translator components are hosted. To our knowledge, transparent shaping is the only application integration technique exhibiting both features. That said, transparent shaping is intended to complement, rather than compete with, the approaches in the second category. Specifically, we plan to employ the automatic translation techniques provided by those approaches in our future work.

9.7 Summary

In this chapter, we have demonstrated how transparent shaping can be used to facilitate transparent application integration in the construction of autonomic systems from existing applications. Transparent shaping enables integration of existing applications—developed in heterogeneous programming languages, middleware frameworks, and platforms—through Web services while the integration and self-management concerns are transparent to the original applications. A case study was described, in which we constructed a fault-tolerant surveillance application by integrating existing applications and adding self-management functionality. We used transparent shaping to enable two existing image retrieval applications, one of them developed in .NET and the other in CORBA, to interact as required, without modifying either application directly.

We note that several challenges remain in the domain of transparent application integration, including automatic translation of the semantics of heterogeneous applications and automatic discovery of appropriate Web services. The increasing maturity of business standards, which have supported automated interactions in business-to-business application integration over the past 20 years, addresses these issues to some extent [6]. Examples of electronic businesses based on Web services include ebXML, RosettaNet, UCCNet, and XMethods. Also, the automatic location of services, which is one of the goals of Web services, has been specified in the Universal Description, Discovery, and Integration (UDDI) specification. UDDI is a Web service for registering other Web service descriptions. Together with tools and techniques for transparently integrating existing applications, as described in this chapter, these developments promise to significantly increase the degree to which autonomic computing is used in the Internet.

Further Information. A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at `http://www.cse.msu.edu/sens`. Papers and other results related to the RAPIDware project, including a download of the TRAP/J, TRAP.NET, ACT/J toolkits, and their corresponding source code, are available at `http://www.cse.msu.edu/rapidware`.

Acknowledgments

This work was supported in part by the U.S. 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, and ITR-0313142.

References

1. J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, 41–50, 2003.
2. D. E. Bakken, *Middleware*. Kluwer Academic Press, 2001.
3. W. Emmerich, "Software engineering and middleware: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, 117–129, 2000.
4. A. T. Campbell, G. Coulson, and M. E. Kounavis, "Managing complexity: Middleware explained," *IT Professional, IEEE Computer Society*, 22–28, September/October 1999.
5. A. Gokhale, B. Kumar, and A. Sahuguet, "Reinventing the wheel? CORBA vs. Web services," in *Proceedings of International World Wide Web Conference*, (Honolulu, Hawaii), 2002.

6. S. Vinoski, "Where is middleware?" *IEEE Internet Computing*, 83–85, March–April 2002.
7. S. Vinoski, "Integration with Web services," *IEEE Internet Computing*, 75–77, November–December 2003.
8. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Services Architecture*. W3C, 2004.
9. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana, *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 ed., March 2004.
10. M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, *SOAP Version 1.2*. W3C, 1.2 ed., 2003.
11. S. M. Sadjadi, P. K. McKinley, and 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. To appear.
12. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, 86–93, 2002.
13. S. M. Sadjadi and P. K. McKinley, "Transparent self-optimization in existing CORBA applications," in *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*, (New York), 88–95, May 2004.
14. S. M. Sadjadi, P. K. McKinley, B. H. Cheng, and R. K. Stirewalt, "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.
15. Object Management Group, Framingham, Massachusett, *The Common Object Request Broker: Architecture and Specification Version 3.0*, July 2003.
16. G. A. Cohen, J. S. Chase, and D. Kaminsky, "Automatic program transformation with JOIE," in *1998 Usenix Technical Conference*, June 1998.
17. S. Chiba and T. Masuda, "Designing an extensible distributed language with a meta-level architecture," *Lecture Notes in Computer Science*, vol. 707, 1993.
18. J. Ferber, "Computational reflection in class based object-oriented languages," in *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, 317–326, ACM Press, 1989.
19. AspectDNG Project Summary. Available at `http://sourceforge.net/projects/aspectdng`
20. S. M. Sadjadi and 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.
21. J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.
22. T. Thangarathinam, ".NET remoting versus Web services," Online article, 2003. Available at `http://www.developer.com/net/net/article.php/11087_2201701_1`
23. P. Dhawan and T. Ewald, "Building distributed applications with microsoft .net (ASP.NET Web services or .NET remoting: How to choose)," Online article, September 2002.
24. N. Cheng, V. Berzins, Luqi, and S. Bhattacharya, "Interoperability with distributed objects through java wrapper," in *Proceedings of the 24th Annual*

International Computer Software and Applications Conference (Taipei, Taiwan), October 2000.

25. E. Wohlstadter, S. Jackson, and P. Devanbu, "Generating wrappers for command line programs: the Cal-Aggie Wrap-O-Matic project," in *Proceedings of the 23rd International Conference on Software Engineering*, 243–252, IEEE Computer Society, 2001.
26. A. Sahuguet and F. Azavant, "Looking at the Web through XML glasses," in *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*, 148–159, September 1999.
27. G. C. Gannod, H. Zhu, and S. V. Mudiam, "On-the-fly wrapping of web services to support dynamic integration," in *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, November 2003.
28. W. K. Edwards, *Core Jini*. Prentice-Hall, 1999.