TRANSPARENT SHAPING OF EXISTING SOFTWARE TO SUPPORT PERVASIVE AND AUTONOMIC COMPUTING

By

S. Masoud Sadjadi

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2004

Abstract

Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing

By

S. Masoud Sadjadi

As the computing and communication infrastructure continues to expand and diversify, the need for adaptability in software is growing. Adaptability is especially important to *pervasive computing*, which promises anywhere, anytime access to data and computing resources. The need for adaptation in pervasive computing applications is particularly evident at the "wireless edge" of the Internet, where software in mobile devices must balance several conflicting concerns, including quality-of-service, security, fault-tolerance, and energy consumption. We say that an application is *adaptable* if it can change its behavior dynamically (at run time). Developing and maintaining adaptable software are nontrivial tasks, however. Even more challenging is to enhance existing programs so that they execute effectively in new, dynamic environments.

We propose a new programming model called *transparent shaping*, which supports dynamic adaptation in existing programs. The key insight in transparent shaping is the synergy resulting from the integration of four key fundamental technologies: *aspect-oriented programming* to enable separation of concerns at development time, *behavioral reflection* to enable software reconfiguration at run time, *component-based design* to enable independent development and deployment of adaptive code, and *adaptive middleware* to hide the adaptive behavior from the functional code. The major contributions of this dissertation can be summarized as follows.

First, we assess the effectiveness and expressiveness of language support in developing adaptable components separately from the functional code. In a case study, we use

the Adaptive Java language to design and evaluate a component called *MetaSocket*, whose behavior and structure can be adapted at run time in response to external stimuli. We demonstrate how MetaSockets can be used to support adaptation in mobile computing environments.

Second, we investigate how to enhance existing application code *transparently* in order to support dynamic adaptation. We propose transparent reflective aspect programming (TRAP), a development model that enables partial behavioral reflection in existing object-oriented programs. The reflection model provided enables separation of crosscutting concerns at *run time* with minimal overhead.

Third, we demonstrate the use of existing adaptive middleware frameworks to support transparent shaping of distributed applications. As a proof of concept, we propose the *ACT* framework, which enables new behavior to be added dynamically (and transparently) to running CORBA applications. We demonstrate how ACT can support both adaptation in pervasive computing contexts and interoperability with other middleware frameworks.

Fourth, we assess the potential role of transparent shaping beyond the domain of a single program, specifically to support application integration. We propose several alternative architectures that can be used to integrate heterogeneous applications, while the interoperation is transparent with respect to the applications and distribution middleware.

To my lovely wife, *Leila,* and my dear son, *Parsa*.

Thank you for being patient with me!

ACKNOWLEDGMENTS[1]

My advisor and guidance committee chairperson, Dr. Philip K. McKinley, supervised this work and guided me through this research area. I would like to express my thanks to him for his invaluable advice and the unlimited time he spent to correct my mistakes. Other members of my guidance committee, Dr. Betty H.C. Cheng, Dr. R. E. Kurt Stirewalt, and Dr. Hayder Radha, were always available for all my questions. I would like to thank them for their help and contributions to this work. I am grateful to my colleagues in the Software Engineering and Network Systems Laboratory and in the Computer Science and Engineering Department of Michigan State University for the insightful discussions we had during the course of this research. Especially, I am very thankful to Dr. Laura K. Dillon, Dr. Abdol-Hossein Esfahanian, Khashayar Rohanimanesh, Mohammad Ghavamzadeh, Eric Kasten, Zhinan Zhou, Farshad Samimi, Zhenxiao Yang, Jesse Sowell, Udiyan Padmanabhan, and Laura Campbell.

Last but not least, I would like to thank my family: my wife, Leila, who encouraged me to start this PhD program; my son, Parsa, who motivated me to graduate; my dear uncles, Ali Sajadi and Hossein Sajadi, who supported me all along; and my parents, Ahmad Sadjadi and Fakhrol-Molook Modarresi, who have always been there for me.

[1]This work has been 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.

TABLE OF CONTENTS

LIST OF FIGURES **x**

1 Introduction **1**

2 Background and Related Work **9**
2.1 Traditional Middleware 10
2.2 Key Supporting Techniques for Adaptation 12
2.2.1 Computational Reflection 13
2.2.2 Component-Based Design 14
2.2.3 Aspect-Oriented Programming 16
2.2.4 Software Design Patterns 17
2.3 A Taxonomy of Adaptive Middleware 18
2.3.1 Middleware Layer 18
2.3.2 Middleware Access Type 20
2.3.3 Middleware Composition Time 22
2.4 Adaptive Middleware Examples 24
2.4.1 ACE, TAO, and Relatives 25
2.4.2 DynamicTAO and UIC 27
2.4.3 QuO 29
2.4.4 Open ORB and Cousins 31
2.4.5 Infopipes and Squirrel 33
2.5 Toward Transparent Shaping 34

3 Designing Adaptable Components **35**
3.1 Adaptive Java Background 37
3.2 MetaSocket Design and Implementation 39
3.2.1 Internal Architecture and Operation 40
3.2.2 Syntax of Absorption and Metafication 43
3.3 Case Study: Adapting an Audio Streaming Application 45
3.3.1 ASA Architecture and Operation 46
3.3.2 Filter Components 49
3.4 Performance Evaluation 53
3.4.1 Adapting to Simulated Packet Loss 53
3.4.2 Adapting to Real Packet Loss 55
3.5 Related Work 57
3.6 Summary 59

4 Transparent Shaping of Object-Oriented Programs **61**
4.1 Background 63
4.1.1 Behavioral Reflection and Java 63
4.1.2 Aspect-Oriented Programming and Java 63

4.2 TRAP/J Operation 64
4.2.1 Overview 64
4.2.2 TRAP/J Run-Time Model 65
4.3 Case Study: Transparent Shaping of ASA 69
4.3.1 Example Application 69
4.3.2 Making ASA Adapt-Ready 70
4.3.3 Adapting to Loss Rate 75
4.3.4 Balancing QoS and Energy Consumption 76
4.4 Related Work 78
4.5 Summary 81

5 Transparent Shaping of CORBA Programs 83
5.1 Background 85
5.1.1 CORBA 85
5.1.2 CORBA Portable Request Interceptors 87
5.2 ACT Architecture and Operation 88
5.2.1 ACT Core Components 89
5.2.2 Interaction among ACT Components 92
5.3 Generic Proxy 94
5.3.1 Architecture 94
5.3.2 Operation 95
5.4 ACT/J Implementation 95
5.5 Case Study: Transparent Self Optimization 97
5.5.1 The Example Application and Experimental Environment 98
5.5.2 Self-Management and Self-Optimization 99
5.6 Case Study: Coupling ACT and QuO 102
5.6.1 QuO Background 103
5.6.2 Dynamic Weaving of Qoskets Using ACT 105
5.6.3 Example: Supporting Unanticipated Adaptation 105
5.6.4 Experimental Results 107
5.7 Related Work 111
5.8 Summary 114

6 Transparent Application Integration 115
6.1 Web Services Background 116
6.2 Transparent Shaping and Application Integration 119
6.2.1 Alternative Architectures for Application Integration 119
6.2.2 The Role of Web Services in Application Integration 122
6.3 Case Study: Integrating Two Existing Applications 123
6.3.1 The Image Retrieval Application 124
6.3.2 The Sample Grabber Application 126
6.3.3 Application Integration Strategy 128
6.3.4 Exposing the Frame Grabber Application as a Web Service 129
6.3.5 Transparent Shaping of the Image Retrieval Client Program 135
6.4 Related Work 138

6.5 Summary 142

7 Conclusions 144

7.1 Contributions 144
7.2 Achievements 147
7.3 Future Work 148

APPENDICES 152

A Transparent Integration of Heterogeneous Distributed Applications 153

A.1 A Stock Quote Web Service 153
A.2 Transparent Integration of Java RMI Applications and Web Services 156
A.2.1 Enabling Java RMI Client Applications to Use Web Services 156
A.2.2 Exposing Java RMI Server Applications as Web Services 162
A.3 Transparent Integration of CORBA Applications and Web Services 164
A.3.1 Enabling CORBA Client Applications to Use Web Services 164
A.3.2 Exposing CORBA Server Applications as Web Services 167
A.4 Transparent Integration of .NET Remoting Applications and Web Services . . . 168
A.4.1 Enabling .NET Client Applications to use Web Services 171
A.4.2 Exposing .NET Server Applications as Web Services 172

BIBLIOGRAPHY 174

List of Figures

1.1 A transparent shaping design tree illustrating a family of adaptable programs produced from an existing program, which is the root of this tree. Children of the root are adapt-ready programs. Other descendants are adaptable programs. 5

2.1 Relationship between meta-level objects and base-level objects. 13

2.2 Component-based design enables static composition and dynamic recomposition. 15

2.3 Conceptual representation of aspect-weaving. (Adapted from [1].) 17

2.4 Middleware services decomposed into four layers, defined by Schmidt [2]. . 19

2.5 Incorporating middleware using integration method. 20

2.6 Incorporating middleware using interception method. 21

2.7 Middleware type according to the time of incorporating adaptive behavior. . 23

2.8 State of research in developing adaptable software using adaptive middleware. 25

2.9 ACE architecture [3]. 26

2.10 TAO architecture [4]. 27

2.11 DynamicTAO reified structure [5]. 28

2.12 The UIC personalities [6]. 29

2.13 QuO architecture [7]. 30

2.14 Open ORB reflection model [5]. 32

2.15 Open binding in Open ORB [8]. 32

2.16 Squirrel: dynamic shipping of a smart proxy [9]. 33

2.17 Alternative places to insert hooks for transparent shaping. 34

3.1 Separation of concerns using MetaSockets. 36

3.2 MetaSocket absorption and metafication: (a) Java MulticastSocket as the base-level class; (b) SendMSocket as the base-level component; (c) MetaSendMSocket, a filter-oriented meta-level component. 40
3.3 MetaSocket internal architecture: (a) MetaSendMSocket, a send-only metamorphic multicast socket; (b) MetaRecvMSocket, a receive-only metamorphic multicast socket. 41
3.4 Excerpted code for SendMSocket. 44
3.5 Excerpted code for MetaSendMSocket. 45
3.6 Example of separation of concerns using MetaSockets. 46
3.7 Physical experimental configuration. 46
3.8 Interaction among components in the Audio Streaming Application. 48
3.9 Operation of block erasure code. 50
3.10 Design of forward error correction filters. 51
3.11 Design of packet loss monitoring filters. 52
3.12 MetaSocket performance in an environment with real packet loss. 55
3.13 MetaSocket performance in an environment with real packet loss. 57

4.1 TRAP/J operation at compile time. 65
4.2 TRAP/J run-time support. 66
4.3 A simplified run-time class graph. 67
4.4 TRAP layered run-time model. 68
4.5 Audio streaming in a wireless LAN. 69
4.6 Adaptation strategy. 70
4.7 Excerpted code for the Sender class. 71
4.8 Excerpted generated aspect code. 72
4.9 Excerpted generated wrapper code. 73
4.10 Excerpted generated metaobject code. 75

4.11 The effect of using FEC filters to adapt ASA to high loss rates on a wireless network. 76

4.12 MetaSocket packet loss behavior with dynamic FEC filter insertion and removal. 77

4.13 Trace of energy consumption during experiment using a software measurement technique. 78

5.1 ACT as a template that can be instantiated in different programming languages and can be used to enhance existing adaptive CORBA frameworks. . 84

5.2 CORBA architecture [10]. 86

5.3 A simple CORBA application with request interceptors. 88

5.4 ACT configuration in the context of a simple CORBA application. 89

5.5 ACT core components interacting with the rest of the system. 90

5.6 Request/reply sequence in the client side of an ACT-ready application. . . . 92

5.7 Incorporating generic proxy in the ACT core. 96

5.8 Interceptor Registration Console 97

5.9 Rule Management Console 97

5.10 The configuration of the access points used in the experiment. 99

5.11 An image from the experiment. 100

5.12 Automatic Adaptation Console. 101

5.13 Maintaining the application frame rate using automatic adaptation. 102

5.14 A simplified depiction of the QuO architecture. 104

5.15 Coupling ACT and QuO. 106

5.16 Round-trip delay in ACT/QuO application. 107

5.17 Code for the contract of the new qosket written in CDL. 107

5.18 Excerpted code for the wrapper of the new qosket written in ASL. 108

5.19 Sample image of a monitored instructional laboratory. 108

5.20 Screen capture of a 252 KB version of images displayed in the ACT/QuO application. 109

5.21 Screen capture of a 19 KB version of images displayed in the ACT/QuO application. 110

5.22 Dynamic adaptation in a ACT/QuO hybrid application. 111

6.1 A simplified service-oriented architecture. 117

6.2 SOAP message structure. 118

6.3 Hosting translator components inside a bridge program. 120

6.4 Hosting the translator components inside the requester program. 121

6.5 Hosting the translator components inside the provider program. 122

6.6 Alternative requester-side architectures for application integration through Web services. 123

6.7 Alternative provider-side architectures for application integration through Web services. 124

6.8 Two combinations out of four possible combinations resulting from combining Figures 6.6(a) and 6.6(b) and Figures 6.7(a) and 6.7(b). 125

6.9 A screen dump of the client program GUI showing an aerial photograph of the Sarasota bay. 126

6.10 A screen dump of the sample grabber application GUI. The left panel shows a preview of the live video captured from a video camera and the right panel shows a still image grabbed using the “Grab” toolbar button. 127

6.11 The filter graph of the .NET frame grabber application taken by the GraphEdit tool. 127

6.12 Excerpted code for the Shape program that hosts both the .NET server and frame grabber assemblies. 130

6.13 Direct modifications to the .NET frame grabber application source code. . . 131

6.14 Shape. 133

6.15 The excerpted WSDL description of the sample grabber Web service. 134

6.16 Excerpted code for the condition and action classes for shaping the CORBA image retrieval client program using the ACT framework. 136

6.17 Excerpted code for the Web service translator component defined as a proxy object in the ACT framework. 137

6.18 The slide show IDL file. 138

6.19 Two screen dumps of the CORBA image retrieval client program GUI. One before the application integration and the other after the integration. 138

7.1 Achievements of this dissertation viewed as a whole. 147

7.2 Expanding the set of existing programs supported by transparent shaping. . 149

7.3 Future work on adaptation coordination. 150

A.1 A SOAP request example. 154

A.2 A SOAP response example. 154

A.3 WSDL for net.xmethods.services.stockquote.StockQuote. 155

A.4 A stock quote application developed in Java RMI. 157

A.5 Excerpted code of the aspect generated by TRAP/J to shape the Java RMI client application. 158

A.6 Excerpted code of the reflective classes generated by TRAP/J to shape the Java RMI client application. 159

A.7 Excerpted code of the delegate and proxy classes. 161

A.8 Excerpted code of the proxy class that adds fault-tolerance to the Java RMI client application by using a Web Service if the Java RMI server is not available. 162

A.9 Excerpted code of the Web service. 163

A.10 The stock quote interface defined in IDL. 164

A.11 Implementation of the stock quote server program in CORBA. 165

A.12 Implementation of the stock quote client program in CORBA. 166

A.13 Excerpted code of the rule used in the client program. 167

A.14 Excerpted code of the proxy used in the client program. 168

A.15 Excerpted code of the stock quote object in the .NET remoting application. 169

A.16 Excerpted code of the server program in the .NET remoting application. . . 169

A.17 Excerpted code of the client program in the .NET remoting program. 170

A.18 Excerpted code of the translator component that is hosted inside a bridge program. 171

A.19 A screen dump of the Add Web Service GUI of VisualStudio .NET that adds the reference to the XMethods Web service to a .NET project. 172

A.20 Excerpted code for the Web service written in ASP.NET and hosted inside the IIS Web server. 173

Chapter 1

Introduction

As the computing and communication infrastructure continues to expand and diversify, the need for adaptability in software cannot be overstated. Adaptability is especially important to *pervasive computing*, which promises anywhere, anytime access to data and computing resources with few limitations and disruptions [11–15]. Pervasive computing is becoming a reality by a convergence of the recent advances in electronic technologies, especially wireless communication, and the growth of the Internet. The need for adaptability in pervasive computing is particularly evident at the "wireless edge" of the Internet, where software in mobile devices must balance conflicting concerns such as quality-of-service (QoS) and energy consumption in responding to variability of conditions (*e.g.,* wireless network loss rate). Adaptability is also important to *autonomic computing*, which promises self-managed and long-running systems that require only limited human guidance [16–19]. Autonomic computing supports systems such as financial networks, transportation systems, and water and power systems, which must continue to operate correctly during exceptional situations. Such systems require adaptation in order to survive hardware component failures, network outages, software faults, and security attacks.

Developing and maintaining adaptable software are nontrivial tasks. We say a software application is *adaptable* if it can change its behavior dynamically (at run time) as a response to transient changes in its execution environment (*e.g.*, to address dynamic network conditions) or to permanent changes in its requirements (*e.g.*, to upgrade long-running mission-critical systems). An adaptable application comprises *functional* code, which implements the business logic of the application and supports its imperative behavior, and *adaptive* code, which implements the adaptation logic of the application and supports its adaptive behavior. The difficulty of developing and maintaining adaptable applications comes from the nature of the adaptive code, which tends to *crosscut* the functional code. Example crosscutting concerns include QoS, mobility, fault tolerance, recovery, security, self auditing, and energy consumption. Even more challenging than developing new adaptable applications is enhancing *existing* applications, such that they execute effectively in new, dynamic environments not envisioned during their design and development. For example, many non-adaptive applications are being ported to mobile computing environments where they require dynamic adaptation.

Separation of concerns [20–22] enables the separate development of the functional code from the adaptive code of an application. This separation simplifies development and maintenance, while promoting software reuse. Separation of concerns has become an important principle in software engineering [23], and many development techniques apply it to some degree. Examples include domain-specific languages, generative programming, generic programming, constraint languages, feature-oriented development, and aspect-oriented programming [24]. Presently, the most widely used approach appears to be aspect-oriented programming (AOP) [1, 25, 26]. While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, complicating the development and maintenance of applications. AOP enables these concerns to be isolated from the rest of the application. However, in traditional AOP the adaptive code is tangled with the functional code during compilation. To support adaptable software, the programmer needs a way to maintain this separation at run time.

One way to support separation of concerns at run time is through the use of behavioral reflection [27–29]. According to Maes [30], *behavioral* or *computational* reflection refers to the ability of a program to reason about, and possibly alter, its own behavior. Behavioral reflection enables a system to "open up" its implementation details at run time [31]. A reflective system has a self representation, which deals with the computational aspects (implementation) of the system, and is *causally connected* to the system, meaning that any modifications either to the system or to its representation are reflected in the other. Since the self representation of a reflective system is not tangled with the system, if we incorporate the crosscutting concerns associated with the system as part of its self representation, then the resulting code at run time is not tangled and can be reconfigured dynamically. When combined with AOP, behavioral reflection enables dynamic weaving of cross-cutting concerns into an application at run time [27, 32]. However, mechanisms are needed to enable the dynamic loading and unloading of adaptive code during execution.

Software components are software units that can be independently developed, deployed, and composed by third parties [33]. Well-defined interface specifications supported in component-based design enable adaptive code to be developed independently from the functional code, and potentially by different parties, using the interface as a contract. Component-based design supports two types of composition. In static composition, a developer can combine several components at compile time to produce an application. In dynamic composition, the developer can add, remove, or reconfigure components within an application at runtime. By enabling the assembly of off-the-shelf components from different vendors, component-based design promotes software reuse. When combined with behavioral reflection, component-based design enables a "plug-and-play" capability for adaptive code to be incorporated with functional code at run time that facilitates development and maintenance of adaptable software.

Finally, in many cases it is desirable to hide the adaptive behavior from the application using middleware. Traditionally, *middleware* is intended to mask the distribution of resources across a network and hide differences among computing platforms and networks [34–36]. As observed by several researchers [37–45], however, middleware is also a natural place to incorporate the adaptation required for many different crosscutting concerns. *Adaptive* middleware enables dynamic reconfiguration of middleware services while an application is running, adjusting the middleware behavior to environmental changes dynamically.

This dissertation proposes a new programming model, called *transparent shaping*, which supports the design and development of adaptable programs from existing programs without the need to modify the existing programs source code directly. The key insight in transparent shaping is the synergy resulting from the integration of four key fundamental technologies: *aspect-oriented programming* to enable separation of concerns at development time, *behavioral reflection* to enable software reconfiguration at run time, *component-based design* to enable independent development and deployment of adaptive code, and *adaptive middleware* to encapsulate the adaptive code inside middleware.

Adaptable programs derived from an existing program share the business logic of the existing program and differ only in their adaptive behavior. Because of such commonality, instead of developing each adaptable program individually, transparent shaping provides a model to produce a *family* of adaptable programs from an existing program. A *program family* [46] is a set of programs whose extensive commonalities justify the expensive effort required to study them as a whole rather than individually.

An adaptable program produced by transparent shaping comprises the original program code, which is fixed during the program execution, and an adaptive code, which can be replaced with another adaptive code dynamically. Replacing one adaptive code with another adaptive code converts an adaptable program into another adaptable program in the corresponding family. This conversion is possible, since the adaptive code is not tangled into the functional code. We use the term *composer* to refer to the entity that performs this conversion. The composer might be a human – a software developer or an administrator interacting with a running program through a graphical user interface – or a piece of software – a dynamic aspect weaver, a component loader, a runtime system, or a metaobject.

As illustrated in Figure 1.1, transparent shaping produces adaptable programs in two steps. 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 program whose behavior can be adapted at run time by a composer inserting or removing adaptive code at certain points in the execution path of the program, called *sensitive joinpoints*. To support such functionality, in the first step, transparent shaping weaves generic interceptors, called *hooks*, at the sensitive joinpoints, which may reside inside the program code itself or inside its supporting middleware. Example techniques for implementing hooks include aspects at compile time, CORBA portable interceptors [47] at startup time, and byte-code rewriting [48] at load time. In the second step, the hooks in the adapt-ready program are used by the composer to convert the adapt-ready program into an adaptable program, as need arises. Adapt-ready programs derived from the same existing program are different in their corresponding sensitive joinpoints and hooks.

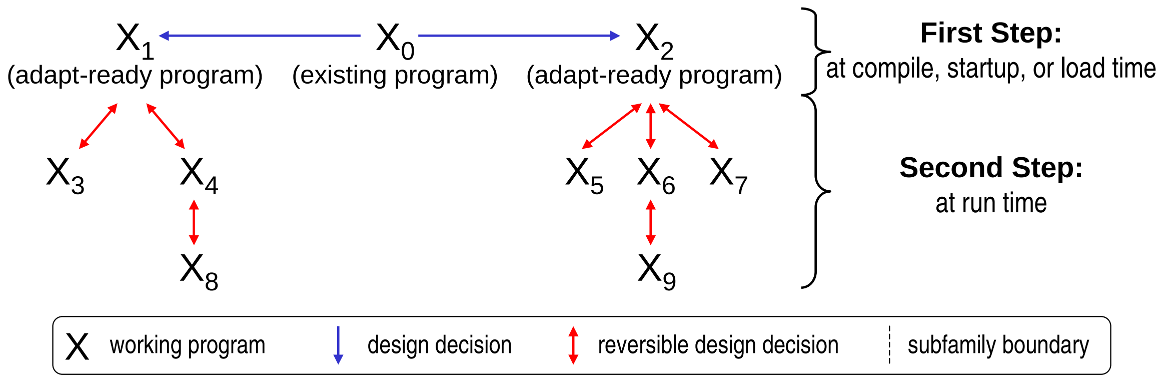


Figure 1.1: A transparent shaping design tree illustrating a family of adaptable programs produced from an existing program, which is the root of this tree. Children of the root are adapt-ready programs. Other descendants are adaptable programs.

As an example, let us consider an existing distributed program (X_0) originally developed for a wired and secured network. To enable this program to run efficiently in a mobile computing environment, transparent shaping can be used to produce an adapt-ready version of this program (X_1), which has hooks intercepting all the remote interactions (first step). At run time, if the system detects a low quality wireless connection, the composer can insert adaptive code for tolerating long periods of disconnection into the adapt-ready program (producing X_4 from X_1). Later, if the user enters an insecure wireless network, the composer can insert adaptive code for encryption/decryption of the remote interactions into the program (producing X_8 from X_4). Finally, when the user returns to an area with a secure and reliable wireless connection, the composer can remove the adaptive code for both security and connection-management to avoid unnecessary performance overhead resulted from the adaptive code (producing X_4 from X_8 and X_1 from X_4).

Thesis Statement. *Transparent shaping provides a programming model for producing adaptable programs from existing programs. Transparent shaping enables reuse of existing programs in new environments even though the specific characteristics of such new environments were not anticipated during the original design of the existing programs.*

The major contributions of this dissertation are summarized as follows.

1. We assessed the effectiveness and expressiveness of language support in developing adaptable components separately from the functional code. Previously, our group developed *Adaptive Java* [49], an extension to Java that contains constructs to support dynamic recomposition. We used Adaptive Java to investigate the process of developing adaptable components [50–54]. Specifically, we designed and evaluated a component called *MetaSocket*, whose behavior and structure can be adapted at run time in response to external stimuli (*e.g.,* wireless channel conditions). Our study shows the use of new constructs in development of adaptable components improves both expressiveness and effectiveness of the adaptive code.

2. We developed a technique to enhance existing application code *transparently* in order to support dynamic adaptation. We proposed transparent reflective aspect programming (TRAP), which enables partial behavioral reflection in existing object-oriented

programs [55, 56]. The reflection model provided enables separation of crosscutting concerns at *run time* with minimal overhead. A prototype of TRAP for Java, called TRAP/J, was developed and used to evaluate TRAP in practice. Results of a case study show the improvement in the execution of an existing application in a mobile computing environment, while the dynamic adaptation is transparent to the application code.

3. We developed an technique to use of adaptive middleware to support transparent shaping of distributed applications. As a proof of concept, we proposed the *ACT* framework, which enables new behavior to be added dynamically (and transparently) to running CORBA applications [57, 58]. In addition, we demonstrated how ACT enables interoperation among otherwise incompatible adaptive CORBA frameworks. The results of evaluating the ACT framework show the overhead introduced by ACT is negligible, while the adaptation provided is highly flexible.

4. To assess the potential role of transparent shaping beyond the domain of a single program, we developed a technique that uses transparent shaping to support application integration. We proposed several alternative architectures that can be used to integrate heterogeneous applications, where the interoperation is transparent with respect to the applications and distribution middleware. As a proof of concept, we used the proposed architectures to provide transparent interoperation in heterogeneous applications developed in Java RMI, CORBA, and .NET Remoting. A case study demonstrates the use of transparent shaping in integration of two existing applications.

The remainder of this dissertation is organized as follows. Chapter 2 provides a background on middleware and adaptation techniques, then introduces a taxonomy of adaptive middleware, and classifies several representative adaptive middleware projects. Chapter 3 introduces MetaSockets. This chapter provides a background on Adaptive Java, describes the internal architecture of MetaSockets, and finally evaluates MetaSockets in a number of case studies. Chapter 4 introduces TRAP and TRAP/J, and shows how TRAP/J can enable existing Java applications to support dynamic adaptation without the need to directly modify their source code. Chapter 5 presents ACT. This chapter provides a background on CORBA and its portable request interceptors, describes the ACT internal architecture, and provides two case studies through which we demonstrate the use of ACT to enhance existing CORBA applications with new adaptive code at run time transparently. Chapter 6 demonstrates the use of transparent shaping in application integration and provides a case study where two existing heterogeneous applications are integrated. Finally, Chapter 7 offers conclusions and discusses future research directions.

Chapter 2

Background and Related Work

Developing distributed applications is a difficult task due to three major problems: the complexity of programming interprocess communication, the need to support services across heterogeneous platforms, and the need to adapt to changing conditions. Traditional middleware (such as CORBA, DCOM, and Java RMI) addresses the first two problems to some extent through the use of a "black-box" approach (*e.g.*, encapsulation in object-oriented programming). However, traditional middleware is limited in its ability to support adaptation. To address all three problems, *adaptive* middleware has evolved from traditional middleware. In addition to the object-oriented programming paradigm, adaptive middleware employs several other key technologies including computational reflection [30], component-based design [33], aspect-oriented programming [1], and software design patterns [59].

Since the transparent shaping programming model benefits from results in adaptive middleware research, in this chapter we review the work in this area. Section 2.1 provides a background on traditional middleware. Section 2.2 describes four key supporting techniques used in the development of adaptive middleware. Section 2.3 proposes a three-dimensional taxonomy that categorizes different adaptive middleware approaches, and describes and compares examples of each. Finally, Section 2.5 discusses different approaches to transparent shaping.

2.1 Traditional Middleware

Middleware is connectivity software that encapsulates a set of services residing above the network operating system layer and below the user application layer, effectively the session and presentation layers of the ISO OSI reference model [35]. Middleware facilitates the communication and coordination of application components that are potentially distributed across several networked hosts. Moreover, middleware provides application developers with high-level programming abstractions, for example, use of remote objects instead of socket programming [36]. In this manner, middleware can hide interprocess communication, mask the heterogeneity of the underlying systems (hardware devices, operating systems, and network protocols), and facilitate the use of multiple programming languages at the application level. Middleware can also be considered as a "glue" that enables integration of different legacy applications [34].

Various paradigms [34, 35] have been used in the development of middleware. Emmerich [35] provides a frequently referenced taxonomy of middleware, which classifies approaches according to four programming-language abstractions used for interaction among distributed software components. *Transactional middleware* [60–62] supports distributed transactions among processes running on distributed hosts. *Message-oriented middleware* [63, 64] facilitates asynchronous message exchange between clients and servers using the message-queue programming abstraction [35]. *Procedural middleware* [65, 66] extends the procedure call in procedural programming languages to include *remote procedure calls (RPC)*, where the body of the procedure resides on a remote host and can be called the same way as a local procedure. Finally, *object-oriented middleware* [67–69] is based on both the object-oriented programming paradigm and the RPC architecture. It provides the abstraction of a *remote object*, whose methods can be invoked as if the object were in the same address space as its client.

Since most research in adaptive middleware is based on the object-oriented paradigm, we focus on that type in more detail. Basically, object-oriented middleware separates object *interfaces*, which comprise a set of functionally related methods, from their *implementations*, which define how objects should respond to messages received from remote objects. Object-oriented middleware also provides a local representation for each remote object, and hides the interprocess communication between a remote object and its local representation. The three major examples of object-oriented middleware are CORBA [67], Java RMI [68], and DCOM/.NET [69, 70]. We review each in turn.

The *Common Object Request Broker Architecture (CORBA)* [47] is a distributed object framework proposed by the Object Management Group (OMG). CORBA supports distributed object-oriented computing across heterogeneous hardware devices, operating systems, network protocols, and programming languages. The *Object Request Broker (ORB)* allows objects to interact transparently with other objects (located locally or remotely). To use a remote object, a client first acquires a reference, called an interoperable object reference (IOR), using either a static file containing the IOR or a CORBA naming service [47]. Next, the client invokes methods on this reference as if the object was located in the client address space. The *Interface Definition Language (IDL)* is a language for defining CORBA interfaces. An IDL compiler generates the code for stubs and skeletons automatically. A *stub* represents a remote object in the client address space and a *skeleton* represents a client in the remote address space. Stubs and skeletons marshal and unmarshal requests and responses to enable object interactions over a network.

Java remote method invocation (Java RMI) [68] was proposed by JavaSoft to support the development of distributed Java-based applications. Java RMI supports distributed computing across heterogeneous hardware devices and operating systems using the Java Virtual Machine (JVM). Unlike CORBA, which is language independent, Java RMI supports only the Java language. Instead of CORBA marshalling and unmarshalling, Java RMI uses object serialization, which preserves the type of the objects being serialized. The registry in Java RMI is similar to a CORBA naming service, which resolves a symbolic name to an actual remote object reference. A server object registers itself with the registry, where a client object can look up the remote object address. Java RMI can dynamically load the class bytecode of an object that is passed between remote objects using Java reflection.

The *Distributed Component Object Model (DCOM)* [69] was proposed by Microsoft as a distributed extension to the Component Object Model (COM) [71]. Similar to CORBA, DCOM supports heterogeneous programming languages, but unlike CORBA and Java RMI, DCOM supports only Windows-based platforms. The DCOM *object proxy* and *object stub* are the equivalent to the CORBA stub and skeleton, respectively. Unlike CORBA and Java RMI, DCOM supports neither multiple inheritance nor exceptions at the IDL level. However, with regard to inheritance, DCOM supports multiple interfaces using a binary standard similar to the C++ vtable [69]. DCOM also supports dynamic invocation using the IDispatch interface [69]. The .NET remoting platform [70, 72] is the follow-on to DCOM. For more detailed comparisons of CORBA, Java RMI, and DCOM/.NET remoting, please refer to [36, 70, 72–74].

2.2 Key Supporting Techniques for Adaptation

In addition to the foundation provided by the design and use of traditional middleware platforms, numerous advances in programming paradigms [1, 23, 30, 33, 59, 75–81] have also contributed to the emergence of adaptive middleware. Although many important contributions have been made in this area [23, 76–81], a review of the literature shows that four paradigms, in addition to object-oriented paradigm, play key roles in supporting adaptive middleware: computational reflection [30], component-based design [33], aspect-oriented programming [1], and software design patterns [59, 75].

2.2.1 Computational Reflection

Computational reflection [30,82] refers to the ability of a program to reason about, and possibly alter, its own behavior. *Reflection* enables a system to "open up" its implementation details for such analysis without compromising portability or revealing the unnecessary parts [31]. In other words, reflection exposes a system implementation at a level of abstraction that hides unnecessary details, but still enables changes to the system behavior [30,82]. As depicted in Figure 2.1, a reflective system (represented as *base-level* objects) has a self representation (represented as *meta-level* objects) that is *causally connected* to the system, meaning that any modifications either to the system or to its representation are reflected in the other [83]. The *base-level* part of a system deals with the "normal" (functional) aspects of the system, whereas the *meta-level* part deals with the computation (implementation) aspects of the system. A *meta-object protocol (MOP)* is a meta-level interface that enables "systematic" (as opposed to ad hoc) inspection and modification of the base-level objects.

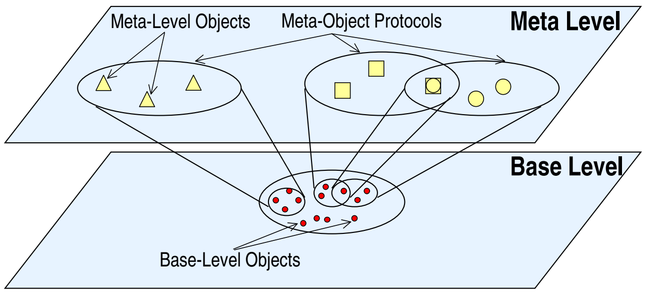


Figure 2.1: Relationship between meta-level objects and base-level objects.

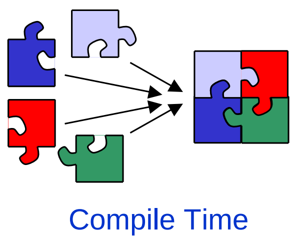
Computational reflection has been studied for several years in the context of programming languages [30, 82, 84–87] and operating systems [88–90]. Recently, reflection has also been studied in middleware, where it enables adapting the behavior of a distributed application by modifying the middleware implementation. Reflective middleware is often concerned with adapting non-functional aspects of distributed applications including QoS, performance, security, fault tolerance, and energy management. Section 2.4 describes several examples of reflective middleware platforms [6, 37, 38, 53, 91–95].

We also note that several reflective programming languages [49, 96–98] have been proposed recently to support development of distributed systems and reflective middleware. *MetaJava* [96] extends Java reflection with behavioral reflection that enables modifying the behavior of the Java RMI package at run time (*e.g.*, encrypting requests before transmitting them over a network). *Program Control Logic (PCL)* [97] provides a programming framework that enables programmers to design, develop, and optimize the performance of adaptive distributed applications [97]. A source-to-source compiler is provided, which inputs meta code specified in a language very close to C++ and Java (PCLC and PCLJ respectively) and outputs a program source in C++ or Java that is then compiled and linked with the base program. *Adaptive Java* [49] is an extension to Java that introduces new language constructs to support behavioral reflection. In its behavioral reflective meta-model architecture, Adaptive Java separates monitoring the behavior (introspection) from changing the behavior (intercession), using “refractive” and “transmutative” meta methods, respectively. *Iguana/J* [98] extends the Java Virtual Machine to intercept method invocation, object creation, and field read and write at run time. Iguana/J can adapt the intercepted operations by loading new code dynamically. These and other reflective languages [30, 82, 84–87] are likely to facilitate the development of adaptive middleware and distributed applications.

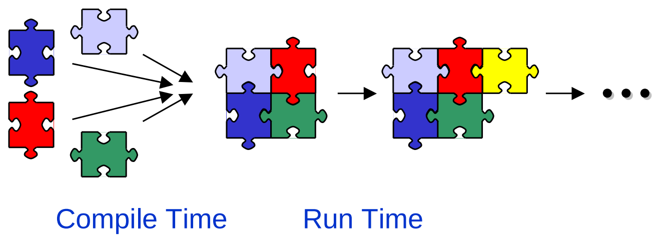
2.2.2 Component-Based Design

Software components are software units that can be independently developed, deployed, and composed by third parties [33]. Components are self-contained: components clearly specify what they require and what they provide. Component-based design (CBD) supports the large scale reuse of software by enabling assembly of “commodity-off-the-shelf” (COTS) components from a variety of vendors [36]. The independent deployment of components enables *late composition* (also referred to as *late binding*), which is essential for adaptive systems. Late composition provides coupling of two compatible components at run time through a well-defined interface. A system developed using CBD is an amalgam of components that can be reorganized easily.

Figure 2.2(a) depicts a *static* composition approach, in which four components are combined at compile time to produce an application. Of particular importance to dynamic recomposition is binding time. *Late*, or dynamic, binding supports coupling of compatible service clients and providers through well-defined interfaces at run time. As shown in Figure 2.2(b), new components can be bound to an application at run time. Moreover, object-oriented languages use *indirect* interfaces, primarily as a means to support inheritance and polymorphism [33]. Effectively, method calls are redirected to the appropriate method implementation. This level of indirection when coupled with dynamic class loading and late binding, helps to support dynamic adaptation.



(a) Static composition.



(b) Static composition and dynamic recomposition.

Figure 2.2: Component-based design enables static composition and dynamic recomposition.

When applied to middleware, CBD provides a flexible and extensible system that can be reconfigured by upgrading each individual component at maintenance time (and possibly at run time) [6, 45, 53, 91, 93, 99, 100]. Specifically, a middleware can be customized to specific application domains, through the integration of domain-specific components, and can evolve using third-party components. Moreover, component-based middleware can be dynamically adapted to its environment using late composition. Examples of major component-based middleware solutions are DCOM [69] (discussed earlier), EJB [101], and CCM [102]. *Enterprise Java Beans (EJB)* [101] is a middleware component model for Java proposed by Sun Microsystems that enables Java developers to use off-the-shelf

Java components, or *beans*. Since EJB is built on top of Java technology, EJB components can only be implemented using the Java language, however. The EJB component model supports adaptation by automatically supporting services such as transactions and security for distributed applications. The *CORBA Component Model (CCM)* [102] is a distributed component model proposed by OMG that can be considered as a cross-platform, cross-language superset of EJB. CCM supports adaptation by enabling injection of adaptive code into component containers (*i.e.*, the component themselves remain intact).

2.2.3 Aspect-Oriented Programming

The third major software development paradigm used in adaptive middleware is aspect-oriented programming (AOP). Kiczales et al. [1] realized that complex programs are composed of different interleaved *cross-cutting concerns* (properties or areas of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, cross-cutting concerns are still scattered among different classes, complicating the development and maintenance of applications. As depicted in Figure 2.3, AOP enables separation of crosscutting concerns during development of the software. Specifically, the code implementing such crosscutting concerns of the system, called *aspects*, are developed separately from other parts of the system. In AOP, locations in the program where aspect code can be woven, called *pointcuts*, are typically identified during development. Later, for example during compilation, an *aspect weaver* can be used to weave different aspects of the program together to form a program with new behavior. AOP proponents argue that disentangling crosscutting concerns leads to simpler development, maintenance, and evolution of software [1,22]. Examples of AOP approaches include AspectJ [103], Hyper/J [104], DemeterJ (DJ) [105], JAC [106], Kava [107], PROSE [32], and Composition Filters [78].

These benefits are important to adaptive middleware. AOP enables factorization and separation of cross-cutting concerns from the middleware core [108], which promotes reuse

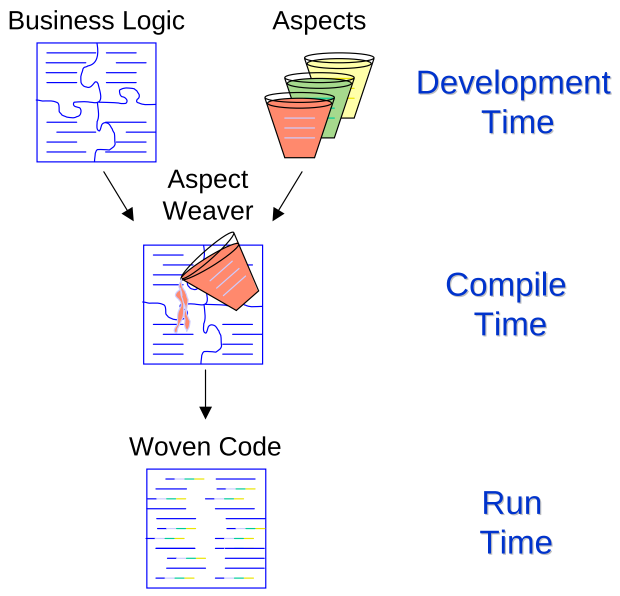


Figure 2.3: Conceptual representation of aspect-weaving. (Adapted from [1].)

of cross-cutting code and facilitates adaptation. Using AOP, customized versions of middleware can be generated for application-specific domains. Yang et al. [109] and David et al. [27] both provide a two-step approach to dynamic weaving of aspects, in the context of adaptive middleware, using a static AOP weaver during compile time and reflection during run time. *PROSE* [32] is an extension to the standard JVM that supports dynamic weaving of aspects into Java programs. Weaving instructions are defined using the JVM debug interface (JVMDI). Other aspect-oriented middleware projects [28, 42, 53, 110, 111] are described in detail in Section 2.4.

2.2.4 Software Design Patterns

Software design patterns [59, 75] provide a way to reuse the software designs that have been successfully used for several years. The goal of software design patterns is to create a common vocabulary for communicating insight and experience about recurring problems and their known “refined” solutions [59].

It is very costly, time consuming, and error-prone to independently rediscover and reinvent solutions to middleware challenges. Schmidt and colleagues [75] have identified a relatively concise set of patterns that enables developing adaptive middleware. For example, the virtual component pattern [112], used in TAO [44] and ZEN [45], enables adapting a distributed application to the memory constraints of embedded devices by providing a small middleware footprint including only a minimum core and a set of “virtual” components, whose code can be dynamically loaded on demand. Numerous adaptive middleware projects [6, 37, 42, 44, 45, 53, 93, 99, 110, 113, 114] benefit from the use of adaptive design patterns, as discussed in Section 2.4.

2.3 A Taxonomy of Adaptive Middleware

Since the Transparent Shaping spans several related concepts in the design of adaptive middleware, we recognized the need to organize the extensive work in this area. Therefore, we have developed a three-dimensional taxonomy, comprising *middleware layer*, *middleware access type*, and *middleware composition time* dimensions, for classifying adaptive middleware projects. The first dimension was introduced by Schmidt [2], while the second and third are proposed by the author.

2.3.1 Middleware Layer

Schmidt [2] decomposes middleware services into four layers: host-infrastructure, distribution, common, and domain-specific services. Figure 2.4 illustrates these layers.

Host-infrastructure middleware resides directly atop the operating system kernel and network protocols and provides a higher-level application programming interface (API) than the ones provided by different operating systems and hides the heterogeneity of hardware platforms, operating systems and, to some extent, network protocols. This middleware layer provides generic services to the upper middleware layers by encapsulating functionality that would otherwise require many tedious, error-prone, and non-portable codes, such as socket programming and thread communication primitives. ACE [113] and Java network package are examples of middleware in this layer.

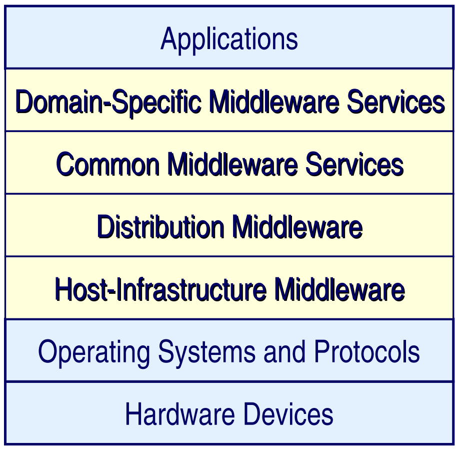

Applications
Domain-Specific Middleware Services
Common Middleware Services
Distribution Middleware
Host-Infrastructure Middleware
Operating Systems and Protocols
Hardware Devices

Figure 2.4: Middleware services decomposed into four layers, defined by Schmidt [2].

Distribution middleware resides atop the host-infrastructure services layer and provides a high-level programming abstraction, such as remote method invocation, to its users. To a great extent, this layer hides the distribution of resources over a network. Using services provided in distribution services layer, application developers do not need to deal with details of network programming (*e.g.*, socket programming), which is a difficult task. CORBA [67], Java RMI [68], and DCOM/.NET [69], discussed earlier, are examples of standard APIs specified in this layer.

Common-middleware services reside atop the distribution layer and support distributed applications with non-functional concerns such as quality of service, fault tolerance, security, load balancing, event propagation, logging, persistence, real-time scheduling, and transactions. The high-level services provided in this layer can be reused in many different applications and are not limited to only a specific domain of applications. QuO [42] is an example of middleware projects in this layer.

Finally, *domain-specific middleware* resides atop the common services layer and is tailored to a specific class of distributed applications. Unlike the common-services layer, the high-level services in this layer can be reused only for a specific domain. Boeing Bold Stroke [115] component-based framework is an example of a proprietary middleware project in this layer.

2.3.2 Middleware Access Type

Studying several middleware projects, we recognized that there are two main methods to incorporating middleware services into distributed applications: integration and interception. We refer to the corresponding middleware services as integrated and intercepting middleware, respectively.

In the *integration* method, a client of a middleware service interacts with the service by sending request messages to the middleware service *explicitly*. In other words, the client is aware of the services provided by the middleware or the client is programmed against the middleware API. A client can be an application program or another middleware service stacked on top of this middleware service. Figure 2.5 shows an application using middleware services by sending request messages to the middleware explicitly. Typically, middleware services adhere to a standard defined in different layers of middleware. CORBA [47] is an example of such a standard in the distribution services layer. TAO [44] and DynamicTAO [37] are examples of adaptive middleware frameworks that adhere to the CORBA standard (*i.e.*, they are CORBA compatible) and provide adaptive real-time services. Using the integrated middleware method, we can develop adaptive middleware that hides details of its adaptive behavior from its users.

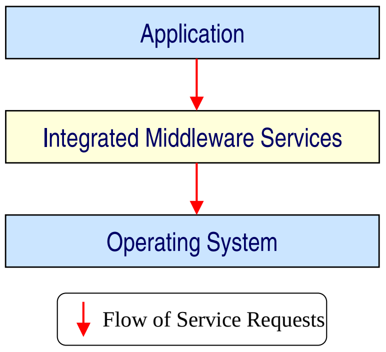


Figure 2.5: Incorporating middleware using integration method.

In the *interception* method, a client application may benefit from the services provided by a middleware service *transparently*. To provide transparency, an intercepting middleware service must capture and handle service request messages originally targeted to either another middleware service or to an operating system service. Among other actions that an intercepting service may perform as a response to an intercepted request, it may reply to the request or it may modify the original request and allow the original target service to reply to the modified request. Figure 2.6(a) illustrates a middleware service intercepting a service request from an application to an operating system. Figure 2.6(b) illustrates a middleware service intercepting a service request from an application to another middleware service. IRL [116] is an example of an intercepting middleware service that provides fault-tolerant CORBA service transparently to the CORBA applications and CORBA implementations. One advantage of interception over the integration method for incorporating adaptive behavior to distributed applications is that the interception method promotes separation of concerns; *i.e.*, non-functional concerns can be developed separately from functional concerns. One disadvantage of interception over the integration method is the overhead that the interception method introduce as a result of extra levels of indirection.

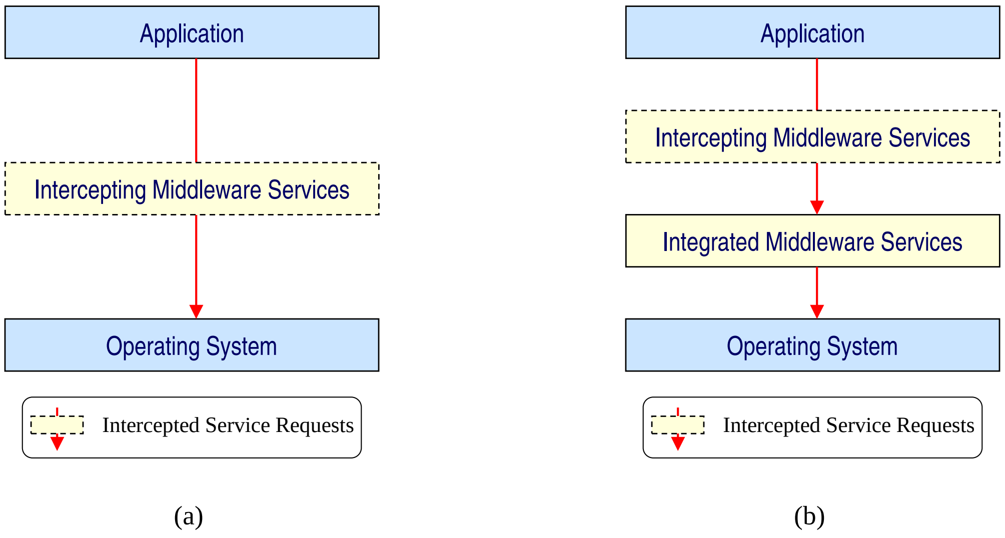

Application
Intercepting Middleware Services
Operating System
Intercepted Service Requests
(a)
Application
Intercepting Middleware Services
Integrated Middleware Services
Operating System
Intercepted Service Requests
(b)

Figure 2.6: Incorporating middleware using interception method.

2.3.3 Middleware Composition Time

As illustrated in Figure 2.7, the lifetime of middleware services can be divided into development time, compile time, startup (or load) time, and run time. Adaptive behavior can be incorporated into middleware at any of these four times. If a middleware approach allows adaptive behavior to be incorporated into the middleware services at development, compile, or startup time, we call it *static* middleware; and if the incorporation of adaptive behavior can continue to run time, we call it *dynamic* middleware.

As illustrated in Figure 2.7, depending on the time adaptive behavior is incorporated into middleware, we identified three classes of static middleware: hardwired, customizable, and configurable middleware. If adaptive code is tangled to the code for middleware services during development time, the middleware is called *hardwired* middleware. Electra [117] is an example of hardwired middleware that incorporates the adaptive code for fault tolerance into its CORBA compliant middleware services. If adaptive behavior is incorporated into middleware services during compile (or link) time, so that a developer can generate customized versions of the middleware services, we call it *customizable* middleware. Please note that the adaptable version is generated in response to the changes realized after the application development time, but before application startup and run time. For example, EmbeddedJava [118] minimizes the footprint of embedded applications during the application compile time. Other examples include approaches that benefit from static weaving of aspects [1] into application source code, compiler flags [45], and precompiler directives [45]. If the incorporation of adaptive behavior into middleware services starts either at development or compile time and ends at startup time, we call it *configurable* middleware. Typically, the interception facilities are integrated with the functional code at development time (*e.g.,* QuO [42]). Alternatively, the interception facilities can be woven into the functional code at compile time(*e.g.,* using AspectJ [103]) or at startup time (*e.g.,* using a configuration file in ORBacus [119] or a command line argument in JacORB [120]). Eternal [121], IRL [116], and Rocks [122] are among examples of configurable evolution approaches.

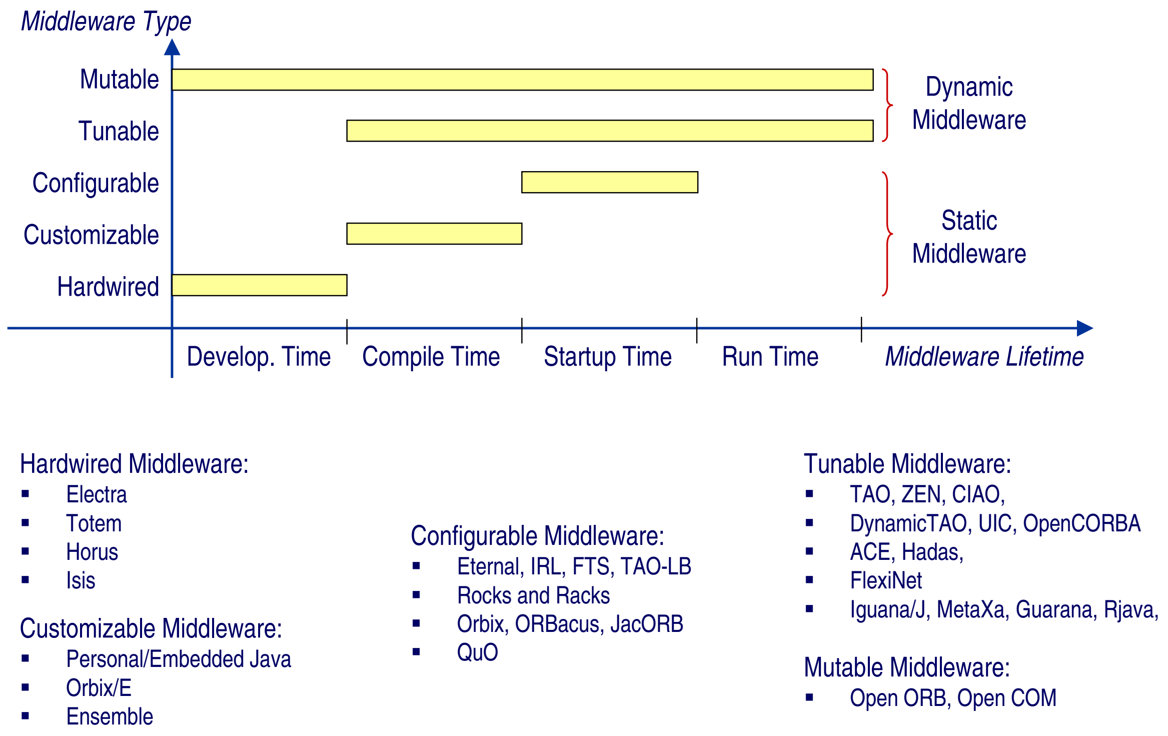


Figure 2.7: Middleware type according to the time of incorporating adaptive behavior.

As illustrated in Figure 2.7, depending on the starting time that adaptive behavior is incorporated into middleware, we identified two classes of dynamic middleware: tunable and mutable middleware. If a middleware allows adaptive behavior to be incorporated into middleware services starting at development, compile, or startup time and continuing to run time, we call it *mutable* middleware. Hence, the process of making a program adaptable can be continued while it is being used. Typically, the interception facilities are integrated with the functional code either at development time (*e.g.*, TAO [44]) or at compile time (*e.g.*, OpenCORBA [92]) or at startup time (*e.g.*, Eternal [121]) and using computational reflection and dynamic code loading adaptive code is integrated with the functional code dynamically. In *tunable* middleware, the middleware core services remain intact during the incorporation of adaptive behavior. Hence, mutable middleware in general is capable of evolving an application to something completely different and unexpected, but tunable middleware limits the evolution to only the adaptive code and not the middleware core services. In other words, in mutable middleware there are no middleware core services. Tunable middleware enables fine-tuning of an application in response to the dynamic changes that can be realized only after the application is started. Examples of tunable middleware include the “two-phase” adaptation approaches employed by David et. al [27] and Yang et. al [109], the component configurator pattern [75] used in DynamicTAO [37], and the virtual component pattern [112] used in TAO [44] and ZEN [45]. Examples of techniques used for dynamic middleware include reflection [38], late composition of components [45], and dynamic weaving of aspects [28, 109]. OpenORB [38], also discussed in Section 2.4, is an example of mutable (but not tunable) middleware.

2.4 Adaptive Middleware Examples

Developing adaptable software using adaptive middleware frameworks is an active research area. We summarize the state of this research in Figure 2.8. We note that this figure is not exhaustive and only representative projects are included. This figure shows where we place each project with respect to the middleware layers. The figure also shows what access type is supported by each project (either integration or interception).

In the following discussion, we focus on those projects that are most directly related to our work. A survey of other adaptive middleware projects can be found in [123]. Before beginning our classification of adaptive middleware projects, we should emphasize that a given project may provide services categorized in more than one middleware layer. In such cases, we placed the project in the layer that matches the primary functionality of the middleware project. If a middleware project supports more than one composition time, we say that the project provides a hybrid adaptation. Finally, when related, details of how supporting paradigms used in each project are discussed and compared to other projects.

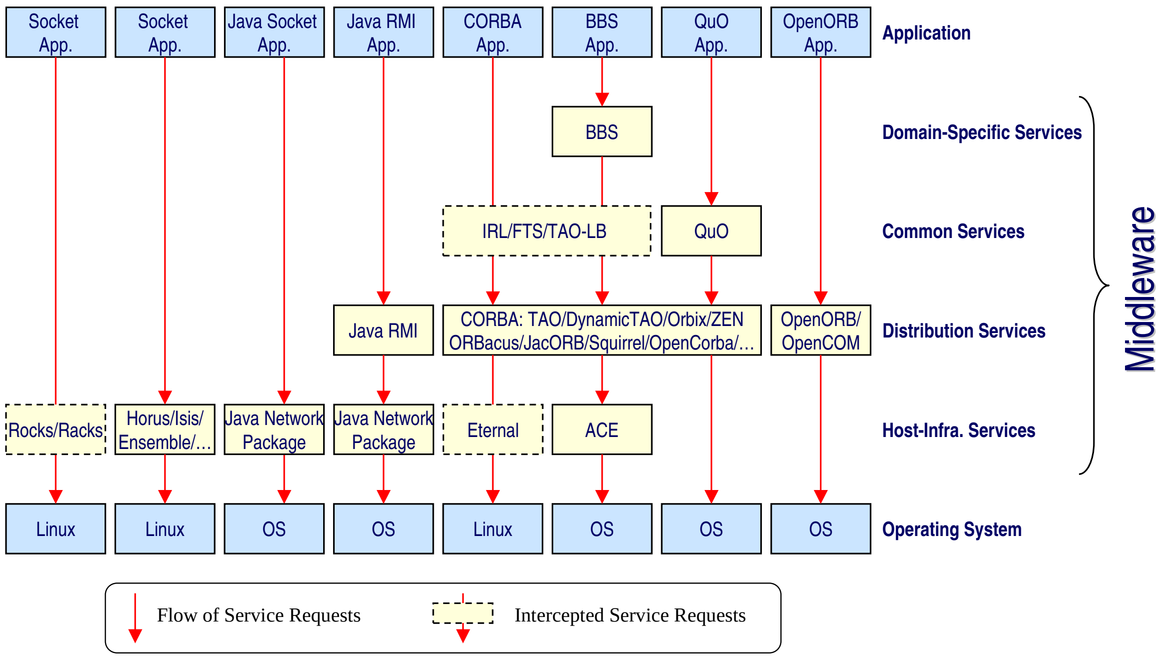


Figure 2.8: State of research in developing adaptable software using adaptive middleware.

2.4.1 ACE, TAO, and Relatives

The distributed object computing (DOC) group has conducted several adaptive middleware projects including ACE [113], TAO [44], CIAO [99], TAO-LB [114], and ZEN [45]. Schmidt's *Adaptive Communication Environment (ACE)* [113, 124] is one of the earliest middleware projects. ACE is a real-time object-oriented communication framework written in C++. ACE employs software design patterns to support distributed applications with efficiency and predictability, including low latency for delay-sensitive applications, high performance for bandwidth-intensive applications, and predictability for real-time applications. Figure 2.9 illustrates the key components in the ACE framework. The *OS Adaptation Layer* resides directly atop the native operating system APIs, providing a platform-independent API. Hence, we place ACE in the host-infrastructure layer. ACE components can be dynamically updated using the service configurator pattern [75] and C++ dynamic binding feature. Therefore, we consider ACE as tunable (but not mutable) middleware because the ACE core remains intact during the tuning process. We consider ACE as an integrated middleware because a client of this middleware, for example TAO, must explicitly send request messages to this middleware.

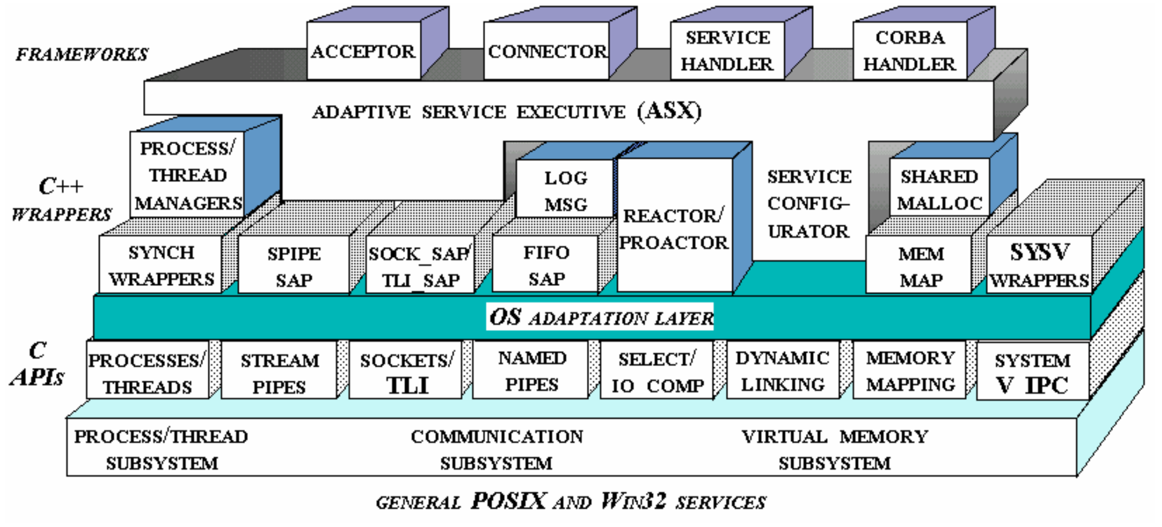


Figure 2.9: ACE architecture [3].

Schmidt et al. [44] extended their ACE work to create *the ACE ORB (TAO)*, a CORBA compliant real-time ORB built atop the ACE components, as shown in Figure 2.10. TAO enhances the standard CORBA event service to provide real-time event dispatching and scheduling required by real-time applications such as avionics, telecommunications and network management systems. Earlier versions of TAO employ the strategy design pattern [59] to encapsulate different aspects of the ORB internals, such as IIOP pluggable protocols, concurrency, request demultiplexing, scheduling, and connection management. A configuration file is used to specify the strategies used to implement these aspects during startup time. TAO parses the configuration file and loads the required strategies. Therefore, we consider TAO as configurable middleware. Recent versions of TAO decomposes the C++ implementation of TAO into several core ORB components that can be dynamically loaded on demand using the virtual component pattern [112]. Therefore, we consider recent versions of TAO as tunable middleware. TAO naturally resides in the distribution layer because it is a CORBA compliant ORB. Similar to ACE, a client of TAO requires to send request messages to used the services provided by TAO. Thus, we consider TAO as an integrated middleware.

The *Component-Integrated ACE ORB (CIAO)* [99], *ZEN* [45], and *TAO load balancing (TAO-LB)* [114] are follow-on middleware projects by the DOC group. CIAO is the TAO implementation of CORBA Component Model (CCM) [47]. ZEN is the TAO implementation in Java and Real-Time Java [125] that provides a micro-ORB architecture. ZEN identifies several major ORB services, such as object adapters and transport protocols, that can be moved out of the micro-ORB kernel. The virtual component pattern [112] is employed to make each service dynamically pluggable. Similar to TAO, CIAO and ZEN are both also considered as integrated middleware. Finally, TAO-LB adds load balancing to TAO [44] transparently from the application code. We consider TAO-LB as an intercepting middleware because it uses the CORBA portable request interceptors to intercept requests messages originally targeted to TAO.

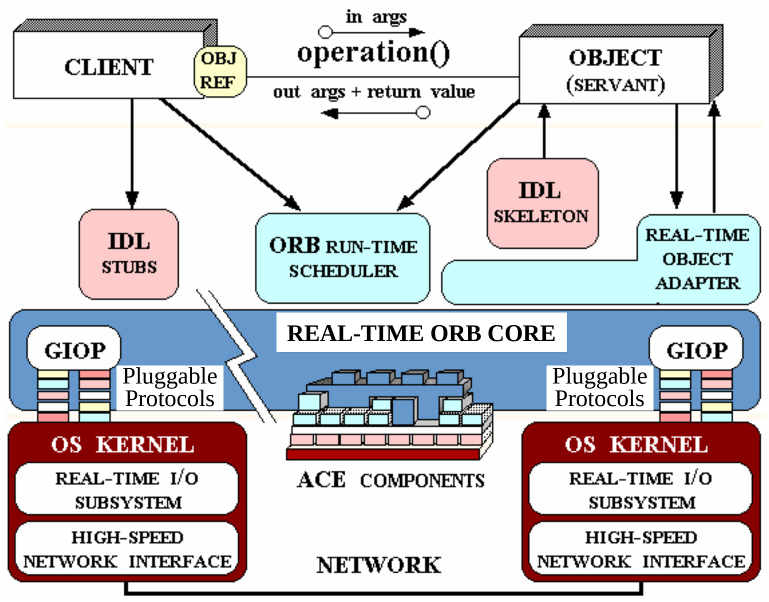


Figure 2.10: TAO architecture [4].

2.4.2 DynamicTAO and UIC

Researchers at the University of Illinois have developed several adaptive middleware platforms [6, 37, 93, 100]. Kon et al. [37] adopted earlier version of TAO [44], which itself is considered as configurable middleware, and built a dynamically adaptive version of TAO called DynamicTAO using computational reflection. To provide real-time services, DynamicTAO uses the Dynamic Soft Real-Time Scheduler (DSRT) [126] that provides QoS guarantees to applications with soft real-time requirements. Reflection in DynamicTAO is achieved using the service configurator pattern [75], which enables configuration and implementation decisions about the ORB services to be deferred until run time. Figure 2.11 illustrates the DynamicTAO reified structure. The DomainConfigurator, TAOConfigurator, and ServantConfigurator are all realizations of service configurator pattern in DynamicTAO. A service configurator in DyanimcTAO exports the DynamicConfigurator interface, which is a CORBA IDL interface, defined also as the MOP for inspecting, adapting, loading, and unloading "component implementations" dynamically. Component implementations are organized in categories representing different aspects of the TAO ORB packaged as dynamically loadable libraries that can be linked to the ORB at run time. We consider DynamicTAO as tunable middleware. Similar to TAO, DynamicTAO is also considered as an integrated middleware.

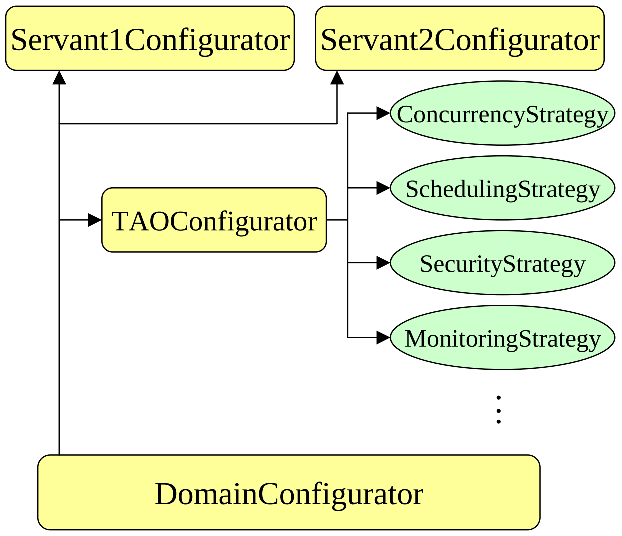


Figure 2.11: DynamicTAO reified structure [5].

Universal Interoperable Core (UIC) [6] is the successor of LegORB [93] both developed at UIUC. UIC, in addition to the small footprint provided in LegORB, can adopt one or more *personalities* such as CORBA, Java RMI, and DCOM for interoperability purposes. Figure 2.12 illustrates the interaction between the UIC core and its personalities. UIC personalities can be either customized statically during the application compile time, or tuned dynamically using late composition of components during run time. UIC minimum ORB core runs uninterruptedly while ORB strategies and servants are dynamically updated. We consider UIC as both customizable and repeatedly-tunable middleware. A UIC client-side ORB for PalmOS can be as small as 16KB. UIC exploits customizable adaptation for the rare and expensive changes during compile time, and exploits tunable adaptation for the frequent and inexpensive changes during run time. Using UIC, the same server objects can interoperate with different personalities without modifying their implementations. UIC naturally resides at the distribution layer. Similar to DynamicTAO, UIC is also considered as an integrated middleware.

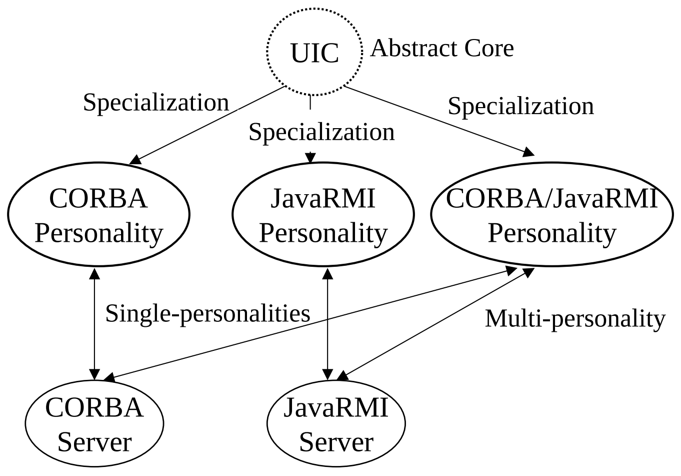


Figure 2.12: The UIC personalities [6].

2.4.3 QuO

Researchers at BBN Technologies have developed an adaptive framework for CORBA and Java RMI applications that supports QoS using aspect-oriented programming paradigm. QuO [42] provides a high-level QoS abstraction at the common-services layer. Figure 2.13 illustrates QuO components residing between the application and distribution ORB. QuO wraps CORBA stubs and skeletons using functional delegates. As illustrated in Figure 2.13, the delegate intercepts outgoing requests and incoming replies. The delegate consults the "contract," using the premethod and postmethod methods. The contract is part of the QuO kernel that is aware of acceptable QoS regions and adapts the application behavior by modifying requests and replies according to the current system status monitored by system condition objects.

QuO provides a quality description language (QDL) to write contracts that specifies QoS regions. The quogen utility can be used to translate these contracts to high-level languages such as C++ and Java. In addition, QuO provides an aspect-oriented structure description language (ASL) that enables developers to write generic or application-specific aspects. Later, the quogen utility can be used to generate delegates from CORBA object interfaces written in IDL, aspects written in ASL, and contracts written in QDL. We consider QuO as customizable middleware because QuO adapts an application during the application compile time using the quogen utility. The delegates in QuO are similar to the statically shipped smart proxies in Squirrel [127] (described later). However, delegates can also wrap skeletons on the server side whereas smart proxies are only at the client side. To use services provided by QuO, a client requires to send request messages to the QuO framework explicitly. Thus, we consider QuO as an integrated middleware.

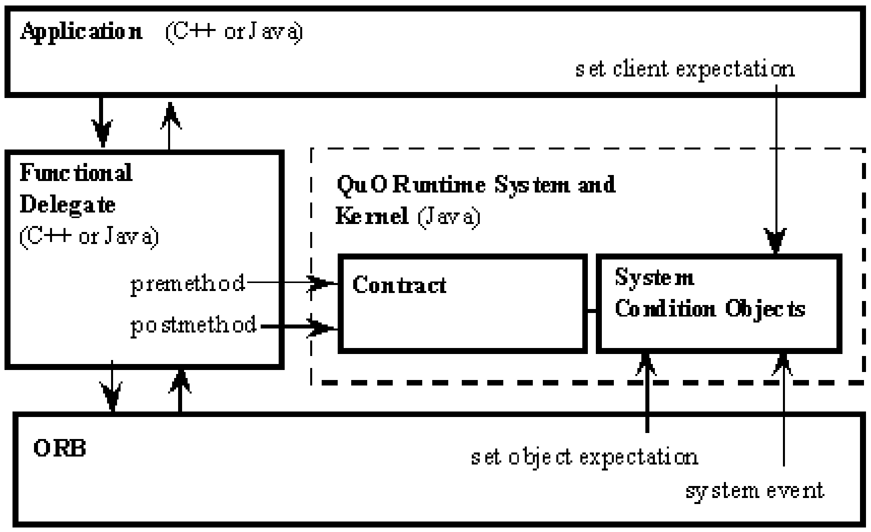


Figure 2.13: QuO architecture [7].

2.4.4 Open ORB and Cousins

Researchers at Lancaster University have conducted several projects in multimedia middleware [38, 83, 128, 129]. Blair et al. [83] have investigated the middleware implementation for mobile multimedia applications which can be dynamically adapted in response to the environmental changes in the context of Adapt project. In the Open ORB project [38], the successor of the Adapt project, Blair et al. continued their investigation studying the role of computational reflection in middleware. More recently, Blair et al. [128] designed Open ORB v2 that adds a component-based design framework to the Open ORB reflective framework. OpenCOM [129] is the implementation of Open ORB v2, designed for Microsoft COM systems. All above mentioned projects are greatly influenced by the ITU-T/ISO RM-ODP [130], a meta standard for multimedia applications. Unlike TAO [44] and DynamicTAO [37], none of Adapt, Open ORB, and Open ORB v2 projects are CORBA compliant.

Open ORB uses reflection to provide dynamic adaptation. The implementation of the Open ORB current reflective architecture is based on the reflection model illustrated in Figure 2.14. Open ORB categorizes reflection into structural and behavioral reflection [5], a distinction first introduced in [131]. *Structural reflection* is the ability of a system to inspect and modify its internal architecture, and *behavioral reflection* is the ability of a system to inspect and modify its computation. Structural reflection is modeled by the “architecture” and “interface” meta-models, and the behavioral reflection is modeled by the “interception” and “resources” meta-models. The *architecture meta-model* provides access to an object using its object graph. The *interface meta-model* provides access to the methods, associated attributes, and inheritance structure of each interface of an object. The *interception meta-model* provides interception hooks for each interface of an object including message arrival, dispatching, marshalling and unmarshalling interception hooks. The *resources meta-model* provides access to available resources per address space and enables resource reservation. Unlike DynamicTAO [37] that uses reflection mainly to implement the service configurator pattern, Open ORB provides an ORB wide reflection. Therefore, we consider Open ORB as mutable middleware. Similar to TAO, Open ORB is also considered as an integrated middleware.

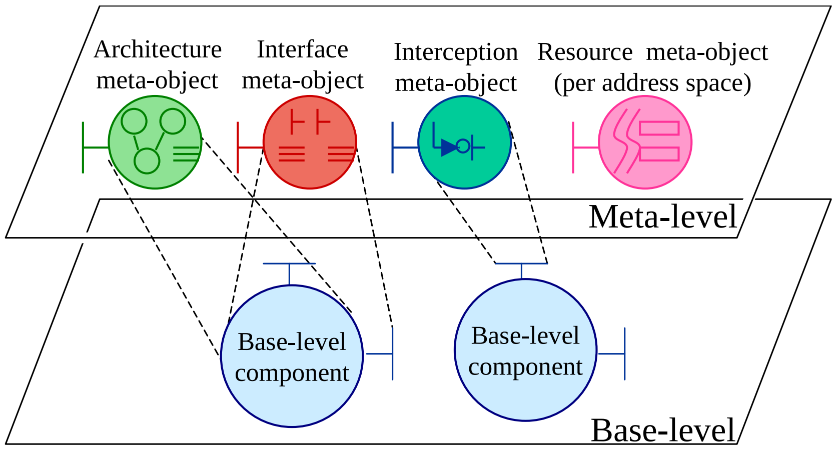


Figure 2.14: Open ORB reflection model [5].

Open ORB supports stream-oriented applications using “explicit binding,” as opposed to the implicit binding provided in CORBA. In explicit binding, remote objects are bound explicitly by a programmer. Figure 2.15 illustrates the result of an explicit binding in a live video application, which represents the end-to-end communication path. Using the Open ORB reflection meta-model (in this case only architecture meta-model), an MPEG encoder can be replaced by an H.263 encoder that uses much lower bandwidth adapting the application to situation that network bandwidth available is decreasing at run time.

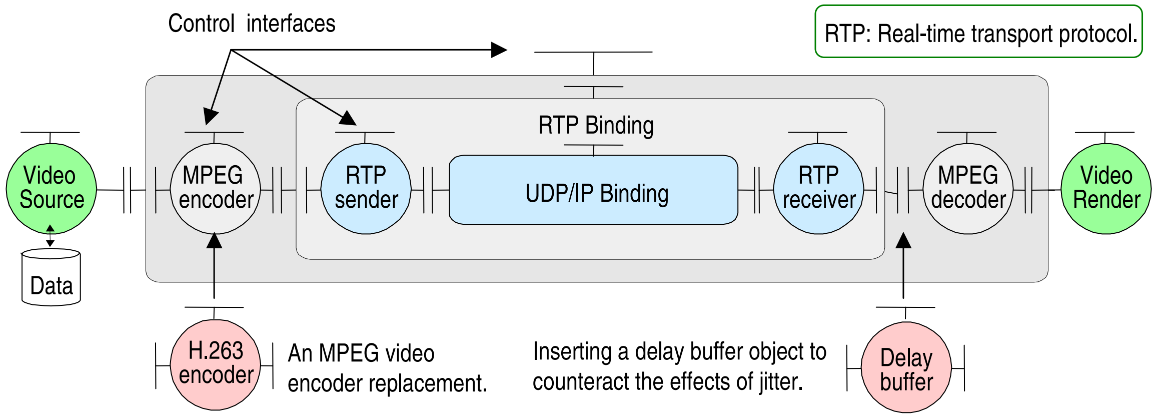


Figure 2.15: Open binding in Open ORB [8].

2.4.5 Infopipes and Squirrel

Infopipes [132], a subproject of Squirrel [127], is a middleware platform for information flow, which is a joint work by University of Kaiserslautern and Oregon Graduate Institute (OGI) base of the Infopipe abstraction [133] jointly introduced with GorgiaTech as part of the Infosphere projects. The designers argue that CORBA stubs and skeletons generated from IDL interfaces follow a standard protocol (marshalling and unmarshalling) that is not suitable for multimedia applications with different QoS requirements. To solve this problem, Squirrel introduces *smart proxies* [9], which are service-specific stubs that include adaptive code. A smart proxy for a specific application can be developed and shipped to the client program statically (during compile time) or dynamically (during run time). Figure 2.16 illustrates dynamic smart proxy shipping in a live video application. We consider Squirrel at the distribution layer because, similar to CORBA stubs, Squirrel uses smart proxies to hide the interprocess communication details from application developers. We consider Squirrel as both tunable and mutable middleware because of its ability to statically and dynamically load smart proxies. The tuning in Squirrel can only occur once at the remote object binding time and the configuration set at the binding time cannot be reconfigured later at run time. For a CORBA object to benefit from the services provided by Squirrel, the object requires to send request messages to Squirrel explicitly. Thus, we consider Squirrel as an integrated middleware.

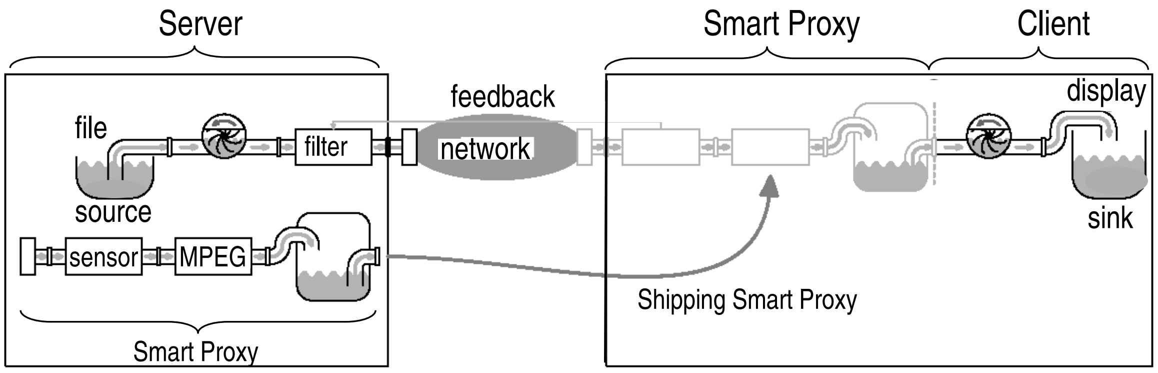


Figure 2.16: Squirrel: dynamic shipping of a smart proxy [9].

2.5 Toward Transparent Shaping

The *transparent shaping* programming model is intended to support production, maintenance, and dynamic reconfiguration of adaptable program families, transparently to their corresponding existing programs. Depending on where the hooks are incorporated inside an existing program during the first step of the shaping process, we identify three approaches to transparent shaping. As illustrated in Figure 2.17, hooks can be incorporated inside an application program itself, inside its supporting middleware, or inside the system platform (network protocols and operating system).

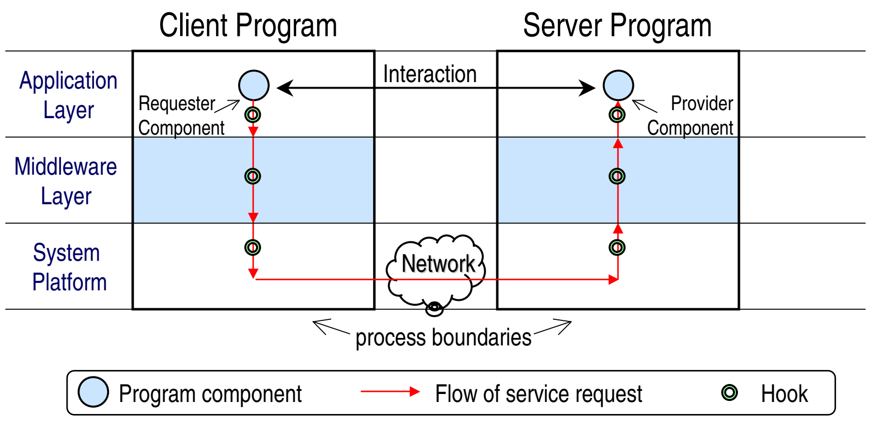


Figure 2.17: Alternative places to insert hooks for transparent shaping.

In this dissertation, we investigate the first two approaches, where the hooks are incorporated either inside the application or inside adaptive middleware. For the former, we propose solutions that weave the hooks inside existing programs using either language extensions or aspect-oriented programming, as described in Chapters 3 and 4, respectively. For the latter, we propose techniques that leverage adaptive middleware mechanisms, such as CORBA portable interceptors [47], as described in Chapter 5. Finally, in Chapter 6, we demonstrate how the combination of these approaches can be used to support a higher level of adaptation, namely, integration of otherwise incompatible applications.

Chapter 3

Designing Adaptable Components

In transparent shaping, the *adaptive code*, which implements the adaptive behavior, must be separately developed from the *functional code*, which implements the business logic of an application. In addition, the adaptive code must be reconfigurable at run time. In an earlier study, our group designed a *composable proxy* [134] in Java that enables mobile Internet users to collaborate via heterogeneous devices and network connections. This approach is based on detachable Java I/O streams, which enable proxy filters and transcoders to be dynamically inserted, removed, and reordered on a given data stream. Using this adaptable component, proxy services can be reconfigured dynamically. Although the resulting system performed well, our experience in this study showed that developing adaptive code in object-oriented languages such as Java, which does not provide facilities to design adaptable components, is difficult and error-prone. Also, in some cases we observed that separation of adaptive code from functional code is almost impossible.

In this chapter, we explore the effectiveness and expressiveness of language support in designing adaptable components. Previously, our group developed *Adaptive Java* [49], which extends Java with new constructs and keywords to facilitate the design of adaptable components. Adaptive Java adds behavioral reflection [30] to Java's structural reflection, enabling dynamic reconfiguration of software components. Using Adaptive Java behavioral reflection facilities, we designed and implemented an adaptable middleware component called a MetaSocket. A *MetaSocket* is an adaptable communication component created from existing Java socket classes, but its structure and behavior can be adapted at run time in response to external stimuli such as dynamic wireless channel conditions. Although the socket abstraction is relatively low-level (host-infrastructure services layer), its ubiquity in distributed applications, as well as in middleware platforms, makes it a good place to begin our studies.

A key concept in the MetaSocket model is that adaptive functionality related to communication streams, possibly tangled throughout application code, is extracted and placed in the MetaSocket layer. Application modules and higher-level middleware layers can invoke traditional socket operations using MetaSockets, while the MetaSockets themselves can adapt (or be adapted) to changes in the environment. This separation of concerns, depicted in Figure 3.1, leads to code that is easier to maintain and evolve to incorporate new adaptive functionality.

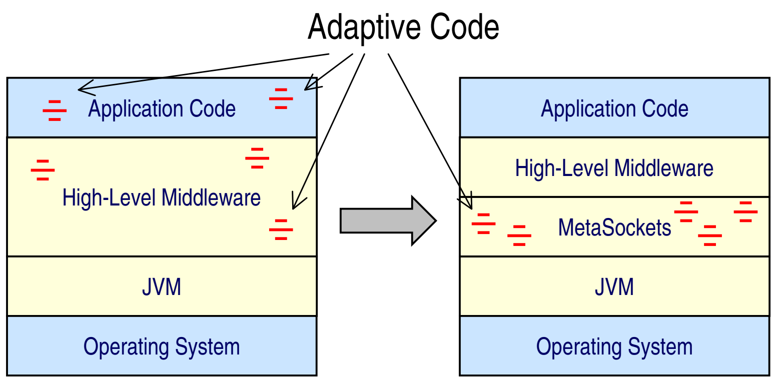


Figure 3.1: Separation of concerns using MetaSockets.

This chapter describes the internal architecture and the operation of MetaSockets and presents a case study in the use of MetaSockets to support audio streaming over wireless channels. The case study, in which iPAQ handheld computers are used as audio “communicators,” illustrates how MetaSockets interact with other adaptive components, such as decision makers and event mediators, to realize run-time adaptability in real-time communication services. The main contribution of this work is to show the effectiveness of programming language support in the development of adaptable software and, through the case study, to reveal several subtle design issues that need to be addressed in the design of such software.

The remainder of this chapter is organized as follows. Section 3.1 provides background information on the Adaptive Java programming language. In Section 3.2, we describe the design and implementation of a MetaSocket variation that is based on the Java MulticastSocket class. Section 3.3 discusses a case study in the use of MetaSockets that supports adaptive error control on wireless audio channels. Section 3.4 presents results of experiments that demonstrate the effectiveness of the proposed methods in adapting to dynamic changes in packet loss rate. Section 3.5 discusses related work, and Section 3.6 summarizes this chapter.

3.1 Adaptive Java Background

Adaptive Java [49] is an extension to Java that adds behavioral reflection to Java's structural reflection, by introducing new language constructs. These constructs are rooted in computational reflection [30, 82], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. A key issue that arises in the application of reflection to middleware platforms is the degree to which the system should be able to change its own behavior. A completely open implementation implies that an application can be recomposed entirely at run-time. In the extreme, all the default components of the system can be destroyed and new ones instantiated, such that the goal of the base-level computation is changed (A spreadsheet can be recomposed as a video player!). On the other hand, limiting adaptability also limits the ability of the system to survive adverse situations.

The basic building blocks used in an Adaptive Java program are *components*, which in this context can be equated to adaptable classes. The key programming concept in Adaptive

Java is to provide three separate component interfaces: one for performing normal imperative operations on the object (*computation*), one for observing internal behavior (*introspection*), and one for changing internal behavior (*intercession*). Operations in the computation dimension are referred to as *invocations*. Operations in the introspection dimension are called *refractions*; they offer a partial view of internal structure and behavior, but are not allowed to change the state or behavior of the component. Operations in the intercession dimension are called *transmutations*; they are used to modify the computational behavior of the component.

An existing Java class can be converted into an adaptable component in two steps. In the first step, a *base-level* Adaptive Java component is constructed from the Java class through an operation called *absorption*, which uses the `absorbs` keyword. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added to and removed from existing components at run time using meta-level transmutations. In the second step, *metafication* enables the creation of refractions and transmutations that operate on the base component. Meta components are defined using the `metafy` keyword. The meta-level can also be given a meta-level (meta-meta-level), which can be used to refract and transmute the meta-level. In theory, this reification of meta-levels for other meta-levels could continue indefinitely [30]. Example code is provided in Section 3.2.2.

Adaptive Java [49] is implemented using CUP [135], a parser generator for Java. CUP takes the grammar productions for the Adaptive Java extensions and generates an LALR parser, called ajc, which performs a source-to-source conversion of Adaptive Java code into Java code. Semantic routines were added to this parser such that the generated Java code could then be compiled using a standard Java compiler.

3.2 MetaSocket Design and Implementation

In this section we describe the architecture and operation of MetaSockets. Our discussion is limited to particular types of MetaSockets designed to enhance the quality of service for multicast communication streams. However, the MetaSocket model is general: MetaSockets can also be used for unicast communication and can be tailored to provide adaptive functionality in other cross-cutting concerns, such as security, energy consumption, and fault tolerance.

Figure 3.2 shows the absorption of a Java MulticastSocket base-level class by a SendMSocket base-level component, and the metafication of this component to a MetaSendMSocket meta-level component. Figure 3.2(a) depicts a Java MulticastSocket class and a subset of its public methods: receive(), send(), close(), joinGroup(), and leaveGroup(). Figure 3.2(b) shows a SendMSocket component, which is designed to be used as a *send-only* multicast socket. The SendMSocket component *absorbs* the Java MulticastSocket class and implements send() and close() invocations that can be used by other components. Other methods of the base-level class are occluded. A link between an invocation and a method indicates a dependency. For example, the send() invocation depends on the send() method, because its implementation calls that method. Figure 3.2(c) shows a MetaSendMSocket component, which metafies an instance of the SendMSocket component and provides a refraction, getStatus(), and two transmutations, insertFilter() and removeFilter(). The use and operation of these primitives will be explained shortly.

In a similar manner, a *receive-only* MetaSocket can be created for use on the receiving side of a communication channel. The RecvMSocket base-level component absorbs a Java MulticastSocket class. In addition to the receive() and close() invocations, this component also provides joinGroup() and leaveGroup() invocations, which are needed for joining and leaving an IP multicast group. All these invocations depend on their respective counterparts in the Java MulticastSocket class. The MetaRecvMSocket metafies an instance of RecvMSocket component and provides the same refractions and transmutations as does the

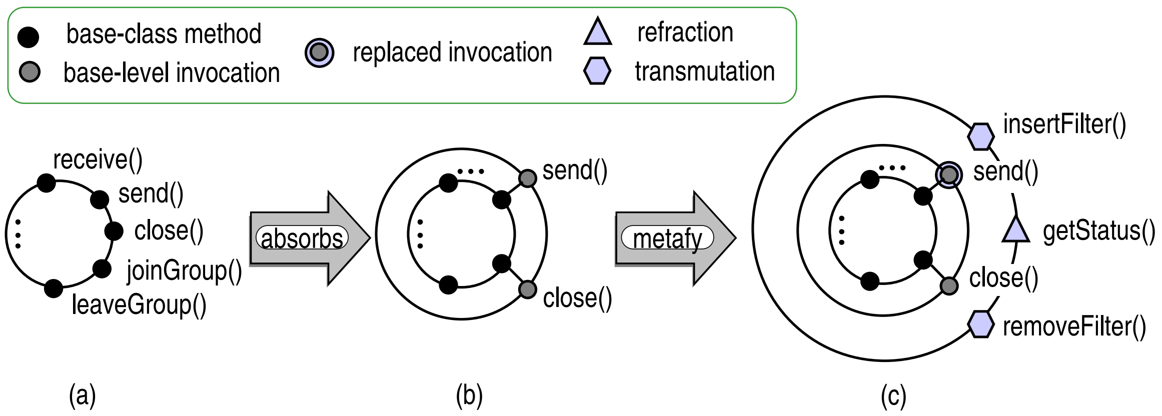


Figure 3.2: MetaSocket absorption and metafication: (a) Java MulticastSocket as the base-level class; (b) SendMSocket as the base-level component; (c) MetaSendMSocket, a filter-oriented meta-level component.

MetaSendMSocket component. The code for MetaSendMSocket and MetaRecvMSocket can be loaded at run time, using the Java Class class and Java reflection package. This dynamic loading of adaptive code enables Adaptive Java applications to adapt to unanticipated changes at run time.

3.2.1 Internal Architecture and Operation

Figure 3.3 illustrates the internal architecture of both a MetaSendMSocket and a MetaRecvMSocket, as configured in our study. In this metafication, packets are passed through a pipeline of Filter components, each of which processes the packet. Example filter services include: auditing traffic and usage patterns, transcoding data streams into lower-bandwidth versions, scanning for viruses, and implementing forward error correction (FEC) to make data streams more resilient to packet loss. In some cases, such as auditing, a filter can act alone on either the sending or the receiving side of the channel. In other cases, such as FEC, modification of the packet stream introduced by a filter on the sender must be reversed by a peer filter on the receiver. In our implementation, when a packet is processed by a filter, an application-level header is prepended to the packet. On the receiver, these headers identify the processing order and filters required to reverse the transformations applied by the sender.

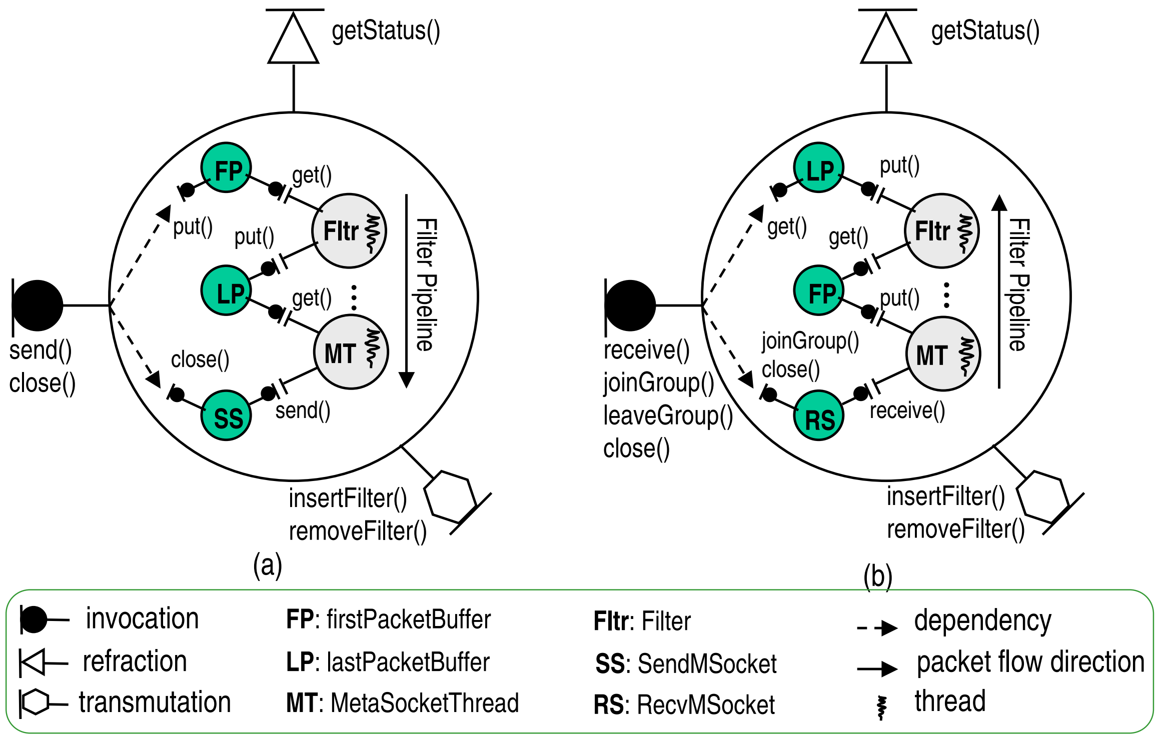


Figure 3.3: MetaSocket internal architecture: (a) MetaSendMSocket, a send-only metamorphic multicast socket; (b) MetaRecvMSocket, a receive-only metamorphic multicast socket.

Packet Buffers. The set of Filter components configured in a MetaSocket pipeline exchange packets via a set of PacketBuffer components. Each filter uses a source and destination packet buffer. Since a packet buffer may be used by multiple threads, its invocations, including get() and put(), are defined as synchronized. All filters in the filter pipeline, execute concurrently, where each filter retrieves a packet from its source packet buffer, processes it, and places it into its destination packet buffer. The destination packet buffer of a filter in the pipeline is either the source packet buffer of the next filter or lastPacketBuffer.

Inserting and Removing Filters. The transmutations insertFilter() and removeFilter() are used to change the filter configuration, and the getStatus() refraction is used to read the current configuration. The insertFilter() transmutation consists of three operations. First, it sets the source packet buffer of the next filter in the pipeline to the new filter's destination packet buffer. Next, it sets the new filter's source packet buffer to the destination packet buffer of the previous filter in the pipeline. Finally, it starts the new filter. The removeFilter() transmutation also consists of three operations. First, it stops the filter that should be removed. Next, it flushes all the packets out of the filter's destination packet buffer and destroys the filter. Finally, it removes the filter from the pipeline and sets the source packet buffer of the next filter to the destination packet buffer of the previous filter in the pipeline. The getStatus() returns a list of filters IDs currently configured in the pipeline.

Sender Operation. Let us consider the sender, as shown in Figure 3.3(a). At the time of metafication, a SendMSocket component is encapsulated by the MetaSendMSocket component. Among other actions, the send() invocation of SendMSocket is replaced by a new send() invocation defined by the meta-level component. After metafication, any call to the base-level send() invocation is delegated to the meta-level send() invocation. This invocation adds a *terminator header* to the datagram packet it receives, which identifies packets that are ready for delivery to the application by the receiver. Next, the meta-level send() invocation stores this packet in firstPacketBuffer (the first packet buffer of the pipeline). Initially, both firstPacketBuffer and lastPacketBuffer refer to the same packet buffer. While lastPacketBuffer may change as new filters are inserted, always pointing to the last packet buffer in the pipeline, firstPacketBuffer remains fixed. When SendMSocket is metafied by MetaSendMSocket, a thread is created and assigned to the SendMSocket send() invocation. This thread loops, retrieving a packet from lastPacketBuffer, creating a datagram packet, and passing it to the original base-level send() invocation, which in turn transmits the packet to the multicast group using the send() method of the underlying MulticastSocket base class.

Receiver Operation. On the receiver, as shown in Figure 3.3(b), a MetaRecvMSocket encapsulates a base-level RecvMSocket component. The receiver can be added to the multicast group, either before or after metafication, by calling its joinGroup() invocation. Once metafied, a thread is assigned to the RecvMSocket receive() invocation. The thread loops continuously, calling receive() and placing the returned packet in firstPacketBuffer. The order of filters on the receiver is the mirror image of that on the sender with function inverted. Each filter in the pipeline processes a packet from its source packet buffer and places it in its destination packet buffer. Similar to the send() invocation on the sender, metafication replaces the base-level receive() invocation with the meta-level receive() invocation defined by MetaRecvMSocket. Instead of calling the RecvMSocket receive() invocation, the MetaRecvMSocket receive() invocation retrieves packets directly from lastPacketBuffer. Before returning the packet to the caller, however, the receive() invocation checks the packet's MetaSocket header. If a terminator header is found at the beginning of the packet, then receive() removes this header and returns the original packet to the caller. Otherwise, additional filter processing needs to be performed on the packet before delivering it to the application. In this case, receive() generates a FilterMismatchEvent event containing the packet and the position of the required Filter in the filter pipeline. (Every Filter at the receiving side performs a similar task and compares the filter ID of the next packet to its ID.) This event is sent to the *EventMediator*, a singleton component in each addresss space that decouples event generators from event listeners [136]. The receive() invocation waits until the event has been handled, meaning that the needed filter has been inserted in the pipeline using the insertFilter() transmutation. Additional details on event handling are discussed in the next section.

3.2.2 Syntax of Absorption and Metafication

Figure 3.4 shows simplified Adaptive Java code for the SendMSocket component. A constructor is defined for this component that creates a new MulticastSocket and sets it as the base-level object for this component (lines 4 to 6). Please note that the base-level object is treated as a secret of the base-level component. A component that uses the SendMSocket component does not necessarily need to know anything about the underlying MulticastSocket or its interface. Two invocations, send() and close() are defined, but they simply call their associated methods from the base object (lines 8 to 12). The code for RecvM-Socket is similar. Once defined, SendMSocket and RecvMSocket can be used via their invocations.

```
 1   public component SendMSocket
 2   absorbs java.net.MulticastSocket {
 3
 4     /* constructor */
 5     public SendMSocket(...) {
 6       setBase(new MulticastSocket(...));}
 7
 8     /* invocations */
 9     public invocation void send(...) {
10       base.send(...); }
11     public invocation void close() {
12       base.close(); }
13   }
```

Figure 3.4: Excerpted code for SendMSocket.

The metafication of these base-level components can be defined at development time or later, at run time. Simplified code for MetaSendMSocket is shown in Figure 3.5. At any point during the execution of the application, a running SendMSocket component can be metafied by calling its constructor (lines 3 to 5). The instance of SendMSocket passed to the constructor of this meta-component is designated as the base-level component. As described earlier, in addition to refractions and transmutations, an invocation, send(), is re-defined in this meta-level component (lines 7 to 9). Defining an invocation at the meta-level is used to replace an invocation of the base-level component. In this example, the new invocation does not call the Java MulticastSocket send() method. Instead, it places the packet in firstPacketBuffer defined as a private field of this meta-component (line 23). Another private field, filterPipeline, is an instance of `java.util.Vector` and keeps track of all the filters currently configured in the MetaSendMSocket (line 22). The refraction getStatus() returns a byte array containing the IDs of these filters (lines 11 to 13). The transmutations insertFilter() and removeFilter() are used to insert and remove filters at specified positions in the filter pipeline (lines 15 to 19). The code for MetaRecvMSocket is similar to that of MetaSendMSocket. In this case, however, the receive() invocation is redefined in the meta-level. In the new definition of this invocation, a packet from the lastPacketBuffer , if available, is delivered to the caller.

```
public component MetaSendMSocket metafy SendMSocket {

  /* constructor */
  public MetaSendMSocket(SendMSocket s) {
    setBase(s); }

  /* replacing the SendMSocket.send() */
  public invocation void send(...) {...
    firstPacketBuffer.put(packet); ...}

  /* refractions */
  public refraction byte[] getStatus() {
    return filterPipeline.getStatus(); }

  /* transmutations */
  public transmutation void insertFilter(int pos, Filter f) {...
    filterPipeline.add(pos, f); ...}
  public transmutation Filter removeFilter(int pos) {...
    return filterPipeline.remove(pos); }

  /* private fields */
  private Vector filterPipeline = new Vector();
  PacketBuffer firstPacketBuffer = new PacketBuffer();
}
```

Figure 3.5: Excerpted code for MetaSendMSocket.

3.3 Case Study: Adapting an Audio Streaming Application

The Java MulticastSocket class is used in many distributed applications. The MetaSockets described in the previous section provide the same imperative functionality to applications and can be used in place of regular Java sockets. In this section, we use an example Adaptive Java application to demonstrate how MetaSockets can further provide adaptive functionality by interacting with other supporting components, such as decision makers and event mediators. A key concept in this approach is that the adaptive functionality, whether it be related to quality-of-service, fault tolerance, or security, is not tangled with the application code. Rather, the "base" application code uses only invocations provided by MetaSockets, while the code that manipulates the behavior of MetaSockets is localized. This separation of concerns, depicted in Figure 3.6, leads to code that is easier to maintain and evolve to incorporate new adaptive functionality. In the following example, we use MetaSockets to support adaptable quality-of-service by reacting to changes in the quality of the wireless channel.

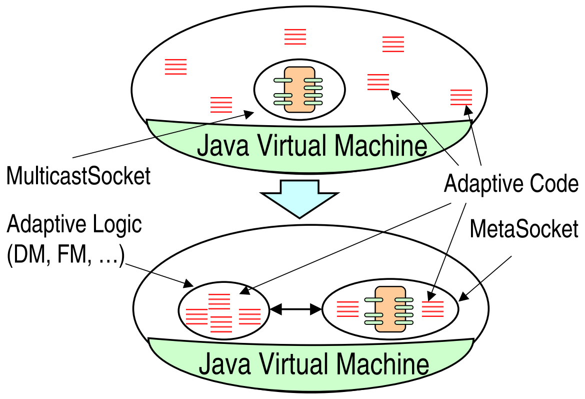


Figure 3.6: Example of separation of concerns using MetaSockets.

3.3.1 ASA Architecture and Operation

In this study, we modified an audio streaming application (ASA) to use MetaSockets instead of regular Java sockets, and we added components to manage the adaptive behavior. We then experimented with ASA by streaming live audio from a desktop workstation to multiple iPAQ handheld computers over an 802.11b wireless local area network (WLAN). The experimental configuration is depicted in Figure 3.7.

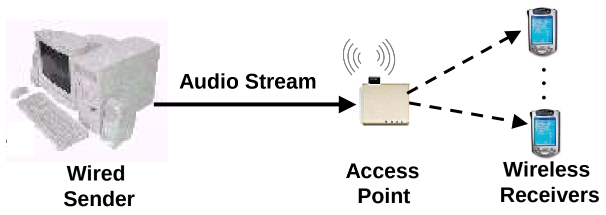


Figure 3.7: Physical experimental configuration.

The ASA code comprises two main parts. On the sending station, the *Recorder* uses the `javax.sound` package to read live audio data from a system's microphone. The audio encoding uses a single channel with 8-bit samples. The Recorder multicasts this data to the receivers via a wireless access point using the send() invocation of a MetaSocket. Each packet contains 128 bytes, or 16 milliseconds of audio data; relatively small packets are necessary to reduce jitter and minimize losses. On each receiving station, the *Player* receives the audio data using the receive() invocation of a MetaSocket and plays the data using the `javax.sound` package.

Figure 3.8 illustrates the major parts of the receiving side of the ASA; the sending side has a similar structure. Please note that we introduce new notations to distinguish the type of interactions among components (One for invocations and another for refractions and/or transmutations). Most of the receiving system executes on an iPAQ handheld computer, but one program, called a *Trader*, executes on a desktop workstation. The two systems communicate over the WLAN. In Adaptive Java, each address space comprises one or more components, each of which in turn may comprise several interacting components. The program running on the iPAQ in Figure 3.8 comprises five main components: a Player, a DecisionMaker, an EventMediator, a ComponentLoader, and a MetaRecvMSocket. The MetaRecvMSocket contains several components that together implement the filter pipeline. As indicated, some of these components are metafied and therefore offer refractive and transmutative interfaces, whereas others are simple base-level components that offer only invocations to other components. The flow of events among components, via an EventMediator, is also shown.

A DecisionMaker (DM) is an optional subcomponent within any Adaptive Java component. According to a set of rules applied to the current situation, a DM controls all of the nonfunctional behavior of the subcomponents of its container component. DMs are arranged hierarchically, such that a given DM inherits rules from a higher-level DM and might provide rules to lower-level DMs. (In our simple example application, the main component on the iPAQ contains a single DM.) Depending on its rules and the current sit-

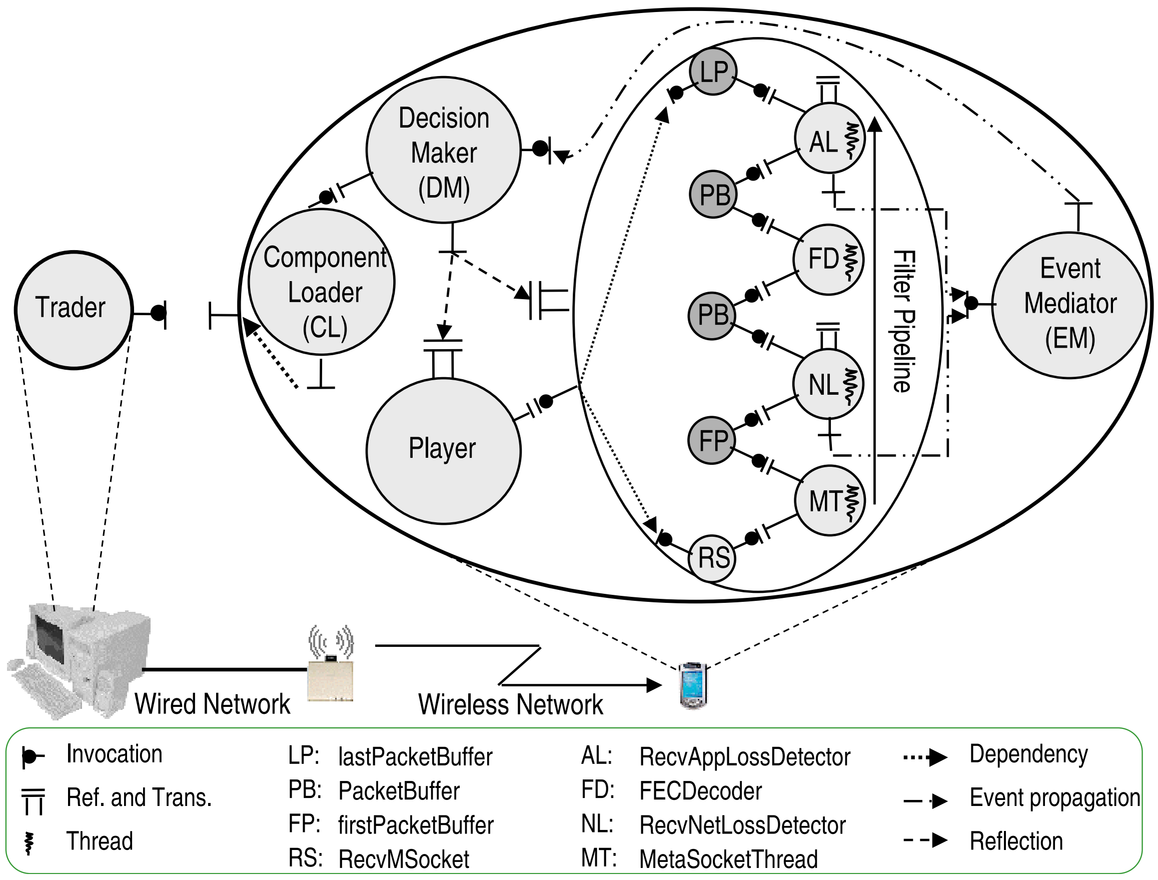


Figure 3.8: Interaction among components in the Audio Streaming Application.

uation, a DM might decide to metafy or change the configuration of an existing component by invoking transmutations of the component. A transmutation might simply set the value of an internal variable, or might involve the insertion or removal of a subcomponent (such as a filter, in our example). In the insertion case, the DM contacts the ComponentLoader (CL) and requests the needed component. The CL is unique to an address space. If the CL does not find the component in its cache, it sends a request to a component *Trader*, which may reside on another computing system. The Trader returns a component implementation corresponding to a syntactic or semantic component request. In our current implementation, we use simple identifiers to search for components. Eventually, the CL uses the `java.lang.ClassLoader` to load this implementation, creates an instance of this class, and returns it to the local DM. The ability to dynamically load components is especially important for mobile devices, where resources might be limited and overhead should be minimized.

Components can interact directly via invocations, refractions and transmutations. To support asynchronous interactions, we implemented an event service. An EventMediator (EM) decouples event generators from event *listeners* [136]. The ASA sender and receiver each contain a single EM that handles all events in the respective program. A listener registers its interest in an event by calling the EM's registerInterest() invocation. When an event is detected by a component, it calls the notify() invocation of the EM. The EM records the event and subsequently alerts all listeners by calling their notify() invocations. To complete the earlier discussion on missing filters, let us consider the situation in which the thread in the receive() meta-level invocation detects that another filter needs to be configured in the pipeline. A FilterMismatchEvent event is sent to the EM, which forwards it to the DM. The DM decides to insert a new filter based on information carried by the event and the pipeline status retrieved using the getStatus() refraction. The DM requests the CL to load the missing filter, after which the DM inserts it at the proper location in the pipeline.

3.3.2 Filter Components

In this case study, we used two types of filters in MetaSockets. The first type provides forward error correction (FEC) encoding and decoding functionality. The second type is used to monitor packet loss conditions and to forward events of interest to the DM. In turn, the DM may decide to insert, remove, or modify an FEC filter.

FEC is widely used in wireless networks, where factors such as signal strength, interference, and antennae alignment produce dynamic and location-dependent packet losses. In current wireless LANs, these problems affect multicast connections more than unicast connections, since the 802.11b MAC layer does not provide link-level acknowledgements for multicast frames. FEC can be used to improve reliability by introducing redundancy into the data channel. Our filters use (n, k) block erasure codes [137]. As shown in Figure 3.9, k source packets are converted into a group of n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [137]. These codes are ideal for wireless multicasting, since a single set of parity packets can correct different packet losses among receivers.

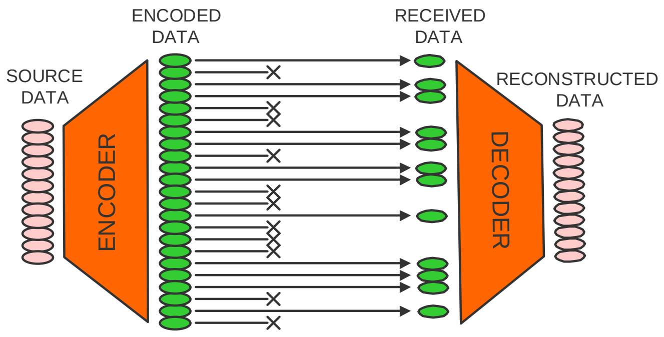


Figure 3.9: Operation of block erasure code.

The FECEncoder and FECDecoder components are extended from the Filter component and use a Java FEC package . The FECEncoder runs on the sender. This component retrieves k packets from its source packet buffer, generates $n - k$ parity packets, and places the original k packets plus the $n - k$ parity packets into its destination packet buffer. The FECDecoder runs at the receiving side and retrieves up to k packets from its source packet buffer, decodes them if possible, and places the recovered original k packets in its destination packet buffer. Any unneeded parity packets are simply dropped. If fewer than k out of the n packets arrive, for a given FEC group, then the FECDecoder retrieves any data packets and places them into its destination packet buffer. The MetaFECEncoder and MetaFECDecoder, shown in Figure 3.10, metafy the FECEncoder and FECDecoder components, respectively. Each provides a getNK() refraction and setNK() transmutation, which are used at run time to read and set the values of n and k. If a packet arrives with a different n or k value than is expected, the MetaFECDecoder fires a FECMismatchNKEvent event. In response, the DM uses setNK() transmutation and adjusts the values for k and n appropriately.

The second type of filter used in our case study monitors events related to packet loss

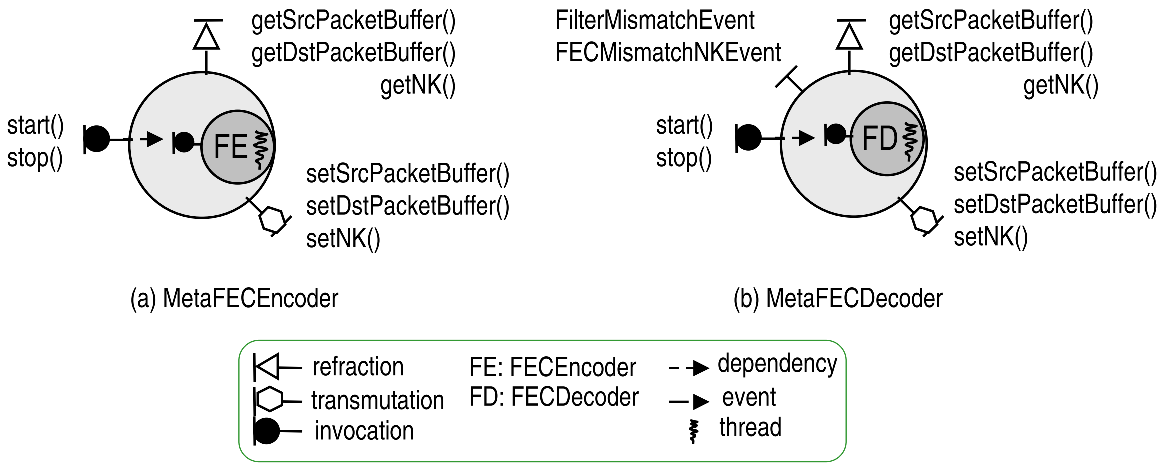


Figure 3.10: Design of forward error correction filters.

rate and reports these to the DM. We developed two sets of filters. The SendNetLossDetector and RecvNetLossDetector filters monitor the raw loss rate of the wireless channel. The SendAppLossDetector and RecvAppLossDetector filters monitor the packet loss rate as observed by the application, which may be lower than the raw packet loss rate due to the use of FEC. The metafied versions of these filters is shown in Figure 3.11. In our experiments, SendAppLossDetector is used as the first filter on the sender side, and RecvAppLossDetector is used as the last filter on the receiver. Conversely, SendNetLossDetector is the last filter on the sender, and RecvNetLossDetector is the first filter on the receiver. The sender's filters simply prepare packets by prepending a header containing the identifier of the corresponding peer filter on the receiver. Each filter on the receiver uses sequence numbers to calculate the packet loss rate over a specified window in the packet stream and stores this information in a vector. Metafying these components provides refractions and transmutations to read the current loss rate and to set or change upper and lower thresholds with respect to the loss rate.

The sender's DM (the global DM) and the receiver's DM (the local DM) work together and use a simple set of rules to make decisions about the use of filters and changes in their behavior. If the loss rate observed by the application rises above a specified threshold, then the global DM can decide to insert an FEC filter in the pipeline or modify the (n, k) pa-

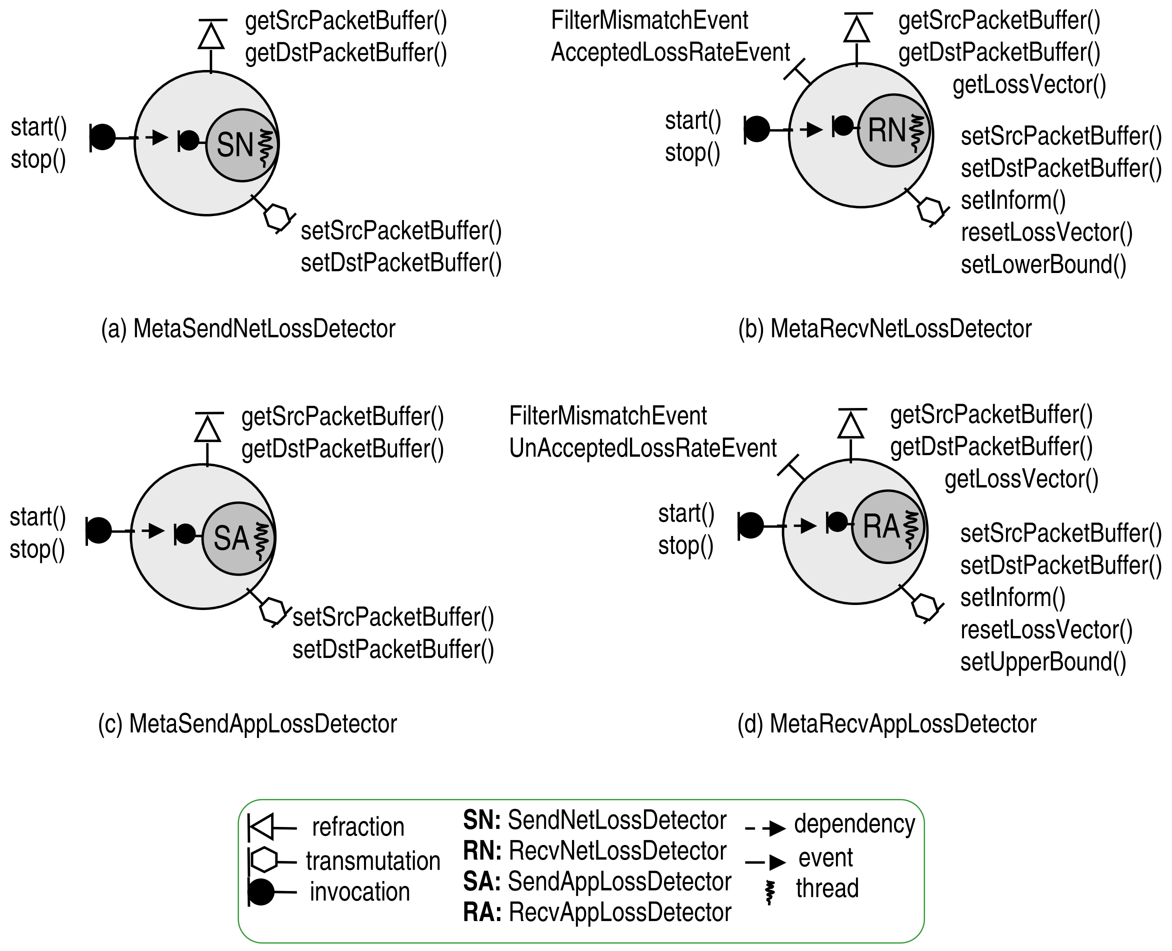


Figure 3.11: Design of packet loss monitoring filters.

rameters of an existing FEC filter. On the other hand, if the raw packet loss rate on the channel drops below a lower threshold, then the level of redundancy may be decreased, or the FEC filter may be removed entirely. To realize this behavior, the local DM uses the setUpperBound() and setLowerBound() transmutations of the metafied filters. The local DM also configures the MetaRecvAppLossDetector to generate an UnacceptableLossRateEvent if the observed loss rate rises too high, by calling the setInform(true) transmutation. When this event fires, the global DM will eventually take action and attempt to reduce the observed loss rate by inserting an FEC filter or changing the parameters of an existing FEC filter. After firing such an event, the local DM calls setInform(false) for the MetaRecvAppLossDetector to suppress further events from this filter. At this time, the local DM also calls setInform(true) for the MetaRecvNetLossDetector, so that an AcceptableLossRateEvent will fire if the network loss rate returns to a satisfactory level. When this event fires, depending on its rules, the global DM can decide to reduce the n-to-k ratio or to remove the FEC filter entirely. As in the first case, the local DM also calls setInform(false) for the MetaRecvNetLossDetector to suppress further events. Any time a filter is inserted or removed on the sender, a FilterMismatchEvent will eventually fire on the receiver, causing the filter pipeline at the receiver to be adjusted accordingly.

3.4 Performance Evaluation

To evaluate the effect of MetaSockets on the performance of audio streaming, we conducted several experiments using ASA. First, we report the effect of using MetaSockets in an environment with simulated packet loss, followed by results with real packet loss on a mobile computing testbed.

3.4.1 Adapting to Simulated Packet Loss

One well-known difficulty in conducting experimental research in wireless environments is the ability to reproduce results, given the highly dynamic nature of the medium [138]. In this set of tests, we created artificial losses by dropping packets in software according to a predefined loss function. In this way, we are able to compare the effects of different parameter settings on the behavior of MetaSockets.

In this experiment, the Recorder program is configured to record 8000 samples per second of live audio, using a single channel at 8 bits per sample. Samples are collected into 128-byte packets packets, that is, each packet contains 16 milliseconds of audio data. We used $(8, 4)$ FEC filters. The upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 30%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 10%.

Figure 3.12 plots packet loss as observed by the two loss monitoring filters on the receiver. The Network Packet Loss curve experiences two periods of high packet loss. The Application Packet Loss curve shows the effect of dynamic insertion and removal of the FEC filter, according to the rules described in Section 3.3.2. When the program begins execution, the sender inserts a SendAppLossDetector filter into its MetaSocket, which quickly causes the receiver to insert the corresponding RecvAppLossDetector. At packet set 8 (meaning the 800th packet), the RecvAppLossDetector filter detects that the loss rate has passed the upper threshold. The filter fires an UnAcceptableLossRateEvent, causing the local DM to request an FEC filter. The global DM decides, based on its set of rules, to insert two filters, an FECEncoder filter with default parameters $n = 8$ and $k = 4$, and a SendNetLossDetector filter, at the second and third positions in the MetaSendMSocket filter pipeline, respectively. When packets containing the headers of the two new filters begin arriving at the receiver, the RecvAppLossDetector detects a packet header that does not match its own identifier. Therefore, it fires a FilterMismatchEvent at two different times, one for each new packet type. These events result in the insertion of a RecvNetLossDetector filter and a FECDecoder filter at the first and second positions in the MetaRecvMSocket filter pipeline, respectively.

As shown in Figure 3.12, the $(8, 4)$ FEC code is very effective in reducing the packet loss rate as observed by the application from packet set 8 to packet set 45. At packet set 45, the RecvNetLossDetector detects that the loss rate has dipped below the 10% lower threshold, so it fires an AcceptableLossRateEvent. In response, the local DM sends a request to the global DM to remove the FEC filter. The DM complies, since under low-loss conditions, the 100% overhead of an $(8, 4)$ FEC code simply wastes bandwidth. It also removes the SendNetLossDetector filter in order to minimize data stream processing under favorable conditions. The arrival of packets without the two headers produces two FilterMismatchEvent events at the receiving side, and the peer filters are removed. As a result, the loss rate experienced by the application is again the same as the network loss rate. At packet set 60, the FEC filter is again inserted, due to high loss rate, and it is later

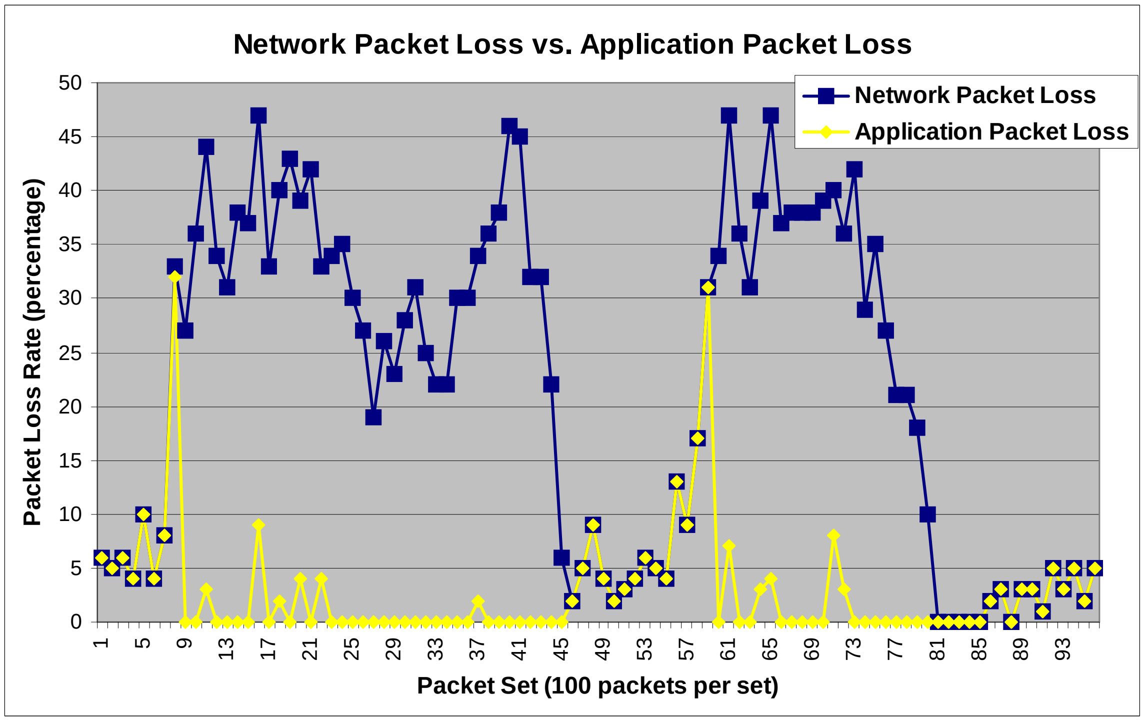


Figure 3.12: MetaSocket performance in an environment with real packet loss.

removed at packet set 80. Considering Figure 3.12 as a whole, we see that the loss rate observed by the application is very low, with the exception of two brief spikes. In order to minimize overhead, FEC is applied only when necessary. This example illustrates how Adaptive Java components can interact at run time to recompose the system in response to changing conditions. While a task such as FEC filter management can be implemented in an ad hoc manner, run-time metafication in Adaptive Java enables such concerns to be added to the system after it is already deployed and executing.

3.4.2 Adapting to Real Packet Loss

Figure 3.13 provides a trace of an experiment, with real packet losses, that demonstrates how MetaSockets adapt to loss rates due to user motion. One user sits at a desktop workstation in our research lab and speaks, while another listens on an iPAQ as he moves about an adjacent hallway. The loss rate is very high while the user is moving. In this particular test, the iPAQ user stood outside the lab for approximately 30 seconds, walked up and down the hall for another 90 seconds, then stood relatively still for another 30 seconds. The upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 10%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 1%. Figure 3.13 plots the packet loss as observed by the two loss monitoring filters on the receiver iPAQ. When the program begins execution, the sending process inserts a SendAppLossDetector filter into its MetaSocket, which quickly causes the receiver to insert the corresponding RecvAppLossDetector. As shown in the Figure 3.13, the loss rate is low at the beginning of the test, then increases quickly when the user starts walking. The RecvAppLossDetector filter detects that the loss rate has passed the upper threshold of 10% and fires an UnAcceptableLossRateEvent. The DM decides, based on its set of rules, to insert two filters, an FECEncoder filter with default parameters ($n = 20$ and $k = 4$ in this particular test), and a SendNetLossDetector filter. When packets containing the headers of the two new filters begin arriving at the receiver, the RecvAppLossDetector detects a packet header that does not match its own identifier. It fires a FilterMismatchEvent at two different times, one for each new packet type. These events result in the insertion of a RecvNetLossDetector filter and a FECDecoder filter in the opposite order as at the sender.

As shown in Figure 3.13, the $(20, 4)$ FEC code is effective in reducing the packet loss rate as observed by the application. The average loss rate in the absence of FEC filters is about 16%, while in the presence of FEC filters the loss rate is improved to 3.5%. Near packet 15,200 the RecvNetLossDetector detects that the loss rate has dipped below the 1% lower threshold, so it fires an AcceptableLossRateEvent. In response, the local DM sends a request to the global DM to remove the FEC filter. The DM complies, since under low-loss conditions, the high overhead of an $(20, 4)$ FEC code simply wastes bandwidth and energy. It also removes the SendNetLossDetector filter in order to minimize data stream processing under favorable conditions. The arrival of packets without the two headers produces two FilterMismatchEvent events at the receiving side, and the peer filters are removed. As a result, the loss rate experienced by the application is again the same as the network loss

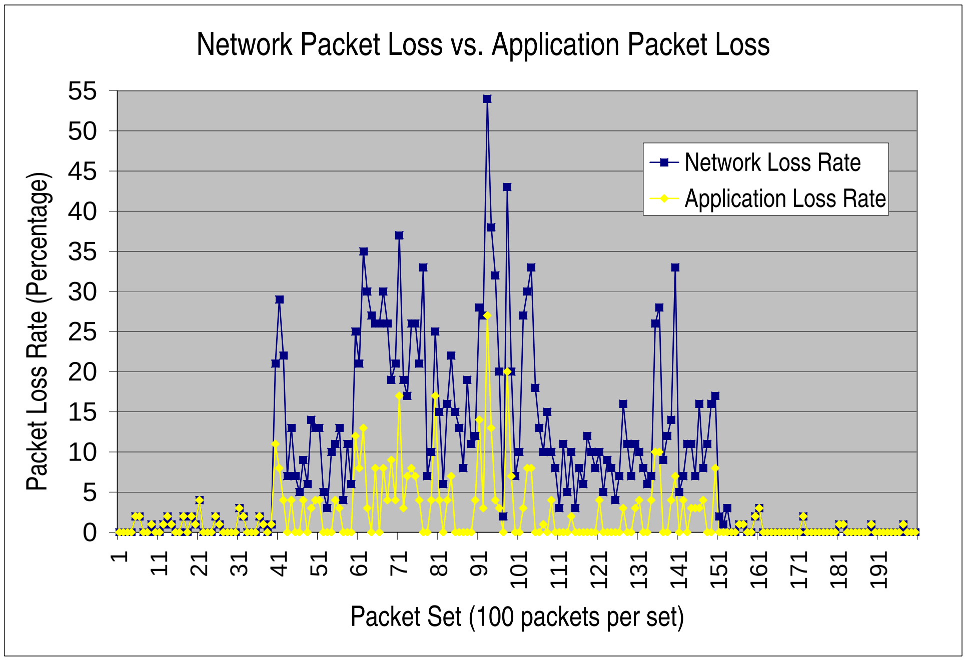


Figure 3.13: MetaSocket performance in an environment with real packet loss.

rate for the remainder of the experiment.

3.5 Related Work

In this section, we identify and discuss three categories of projects related to Adaptive Java and MetaSockets.

The first category includes middleware projects that support adaptive behavior in Java programs by extending the Java Virtual Machine. Examples include Iguana/J [98], Meta Java [96], JDrums [139], Guaraná on Java [140], PROSE [32], and R-Java [141]. A major benefit of implementing adaptation in this way is that the execution of virtually any bytecode instruction can be intercepted within a customized JVM. In contrast, only messages originally targeted for Java sockets can be intercepted and adapted dynamically using MetaSockets. However, some researchers have noted that fine-grained interception at the JVM level can produce significant performance overhead. For example, according to [98], the time for common operations such as creating new objects can be increased by an order of magnitude. Another advantage of JVM-supported adaptation is that it is usually transparent to the target Java program (no code modification required). On the other hand, using a custom JVM tends to limit portability. Since our implementation of Adaptive Java uses source-to-source compilation, MetaSockets can execute atop any standard JVM. Moreover, to address the transparency issue, we developed a generator framework, called TRAP/J, which enables adaptable components such as MetaSockets to be woven into existing Java programs without modifying the application source code. TRAP/J is introduced in the next chapter.

The second category includes projects that use aspect-oriented programming [1] to weave adaptive code into functional code. Although many projects in the AOP community address compile-time weaving [103], a growing number of projects focus on run-time composition [27, 32, 48, 107, 109, 142–147]. By defining a reflection-based component model, Adaptive Java also supports run-time reconfiguration. A related concept is composition filters [78], which provide a mechanism for disentangling the cross-cutting concerns of a software system. This system declares filters that intercept messages received and sent by objects. As such, messages can be massaged and checked before they are delivered to an object, separating aspects, such as security authentication or bounds checking, from the objects that send and receive these messages. Adaptive Java's approach to composition using encapsulation could be used to instantiate a message filtering design where components are extended and invocations added such that a call to an invocation would be filtered through subsequent encapsulation layers. However, such a design would not have the source code expressiveness provided by the declarative specification language in composition filters.

The third category of related work includes projects that, like Adaptive Java, extend the Java syntax and provide new constructs to allow developers to write adaptable applications more expressively. Examples include Open Java [148], FRIENDS [94], PCL [97], R-Java [141], and Handi-Wrap [149]. Open Java provides an approach supporting customized compilers that define new compile-time MOPs [150]. For example, to support writing expressive programs that use a set of design patterns, Open Java enables a developer to build a customized compiler that understands the new syntax. The PCL project [97] also focuses on language support for run-time adaptability. Our concept of "wrapping" classes with base components is similar to the use of *Adaptors* used in PCL. However, modification of the base class in PCL appears to be limited to changing variable values, whereas Adaptive Java transmutations can modify arbitrary structures or subcomponents. Moreover, by combining encapsulation with metafication, Adaptive Java can be used to realize adaptations in multiple meta-levels.

3.6 Summary

As a step toward transparent shaping, in this chapter we studied the effectiveness of new programming language constructs and keywords to separate development of adaptive code from functional code. We used Adaptive Java to develop an adaptable component, called the MetaSocket, which can be used in existing socket-based Java applications. *MetaSockets* can be reconfigured dynamically in response to external stimuli through the use of a filter pipeline. The filter pipeline allows insertion and removal of filters dynamically. Since the core MetaSocket code remains intact during the tuning process, we classify them as repeatedly-tunable middleware (the corresponding taxonomy is introduced in Section 2). The filters in MetaSockets can be developed by third parties and can be independent of the functional code of an application. In other words, MetaSockets provide transparency with respect to adaptive code. The code for MetaSockets is compiled by the Adaptive Java compiler, which is a source-to-source compiler, so the resulting Java program can be compiled by the standard Java compilers and run by the standard JVM.

In a case study, we used MetaSockets to support run time adaptation in iPAQ handheld computers used as audio "communicators." We described in detail how adaptive behavior is implemented and how MetaSockets interact with other adaptive components, including decision makers and event mediators. Results from experiments on a mobile computing testbed demonstrate the effectiveness of these methods in responding to dynamic wireless channel conditions. While this chapter demonstrated the application of MetaSocket to a specific communication service, we emphasize that the Adaptive Java mechanisms are general: any component in the system can be metafied and adapted at run time.

Referring back to the transparent shaping programming model, introduced in Chapter 1 and illustrated in Figure 1.1, Adaptive Java can be used to produce an adaptable program family from an existing Java program. For example, if we start with an existing socket-based Java program, first we can produce an adapt-ready version of this program by modifying all lines in the program that create an instance of any Java socket classes, and then compiling the program using the Adaptive Java compiler. Next, at run time we can insert and remove filters in the adapt-ready program to produce other members of this adaptable program family.

Although MetaSockets proved to be useful in supporting dynamic adaptation, our study of them revealed the following two issues. First, to incorporate a MetaSocket into an existing program, we need to modify the program source code directly, which is not desirable. Second, once the existing program is modified to use a MetaSocket instead of a Java socket, dynamic adaptation is only possible *within* the MetaSocket (*e.g.,* through the insertion and removal of filters). In other words, we cannot replace one version of a MetaSocket with another more appropriate version of the MetaSocket at run time. We address these issues in the next chapter.

Chapter 4

Transparent Shaping of Object-Oriented Programs

In this part of our study, we developed an extension of transparent shaping that can be used to support dynamic adaptation in existing programs developed in class-based, object-oriented programming languages. We call this programming model *Transparent Reflective Aspect Programming (TRAP)*. Unlike Adaptive Java [49], TRAP does not require any direct modifications to the existing programs.

As an extension of transparent shaping, TRAP provide a programming model to produce a family of adaptable programs from an existing program. To enable a developer to balance the flexibility of dynamic adaptation and the performance of adaptable programs, TRAP employs a *two-step* approach to dynamic adaptation. Specifically, TRAP enables the developer to select, at compile time, a subset of classes in the existing program to be reflective at run time. We say a class is *reflective* at run time if its behavior (*e.g.,* the implementation of its methods) can be inspected and modified dynamically. Many class-based, object-oriented languages such as Java and C++ do not support such reflective classes at run time. Therefore, programs developed in these programming languages are required to be modified to accommodate dynamic reflection facilities.

To eliminate the need for direct modifications to an existing program, TRAP uses compile- and load-time program transformation techniques (*e.g.,* compile-time aspect weaving [103], compile-time meta-object protocols [84, 148], and load-time meta-object protocols [48, 147]). Software generator tools produce an adapt-ready version of the program augmented with the required hooks. The hooks provide the reflective facilities for the selected classes required for dynamic adaptation. As the adapt-ready program executes, new behavior can be introduced to the program by insertion and removal of adaptive code via interfaces to the reflective classes.

Extensive use of behavioral reflection in adaptable programs incurs unnecessary overhead and in extreme cases every message sent to an object must be intercepted and possibly redirected [98]. To avoid this problem, TRAP enables a developer to select *what* should be reflective (called spatial selection [151]) at compile time and *when* the reflection should be active (called temporal selection [151]) at run time.

To validate these ideas, we developed TRAP/J, a prototype instantiation of TRAP for Java programs. In this chapter, we focus on the operation of TRAP/J and describe the details of the techniques used to generate adapt-ready programs from existing Java programs, as well as their corresponding subfamily of adaptable programs. In earlier work [109], our group showed how to use aspect-oriented programming to selectively introduce behavioral reflection into an existing program. However, the reflection used there is *ad hoc* in that the developer must invent the reflective mechanisms and supporting infrastructure for adaptation, and must create an aspect that weaves this infrastructure into an existing program. In contrast, TRAP/J employs a *systematic* approach to dynamic adaptation. TRAP/J generates the required reflective infrastructure and weaves it into an existing program automatically.

The remainder of this chapter is organized as follows. Section 4.1 presents background information. Section 4.2 describes the operation of TRAP/J. Section 4.3 presents a case study, where we used TRAP/J to augment an existing audio-streaming application with adaptive behavior, enabling it to operate more effectively across wireless networks. Section 4.4 categorizes related research projects, which address dynamic adaptation in distributed applications, and discusses how TRAP relates to them. Finally, Section 4.5 summarizes this chapter.

4.1 Background

Many approaches to developing adaptable software, including TRAP/J, use behavioral reflection, aspect-oriented programming, or a combination of both. In this section, we explain how these technologies are supported in Java.

4.1.1 Behavioral Reflection and Java

Unfortunately, Java by itself does not support behavioral reflection. Java supports a structural reflection through its `java.reflect` package and the `java.lang.Class` class [152]. The Java reflection facilities enable inspection of a Java program (*e.g.*, to determine the class of a given object, and the methods and fields of that class) as well as to perform limited operations (*e.g.*, to get and set an object's field value and to invoke one of its methods using the method name provided as a string of characters) at run time. Moreover, the `forName()` static method of the `java.lang.Class` allows dynamic class loading in Java programs. Although Java supports structural reflection, it does not support behavioral reflection [30]. TRAP/J and several other projects [27, 49, 98, 107, 109, 140, 141, 148, 149, 151, 153, 154], discussed in Section 4.4, provide behavioral reflection in Java programs.

4.1.2 Aspect-Oriented Programming and Java

AspectJ [103], used in TRAP/J, is a widely used AOP extension to Java. An aspect in AspectJ is a class-like language element, which is used to modularize a crosscutting concern. An aspect has two parts: advice and pointcut. *Advice* is an implementation of a crosscutting concern and a *pointcut* is a set of joinpoints, where the advice is woven. A *joinpoint* is an identifiable point in the execution path of an application, such as a method call or an access to a field. At compile time, a number of such aspects can be selected to be woven into a Java program using the AspectJ compiler, called aspect weaver, to produce a modified version of the program. As described next, TRAP/J uses AspectJ to augment existing

Java programs with the necessary “hooks” to produce adapt-ready versions of the existing programs.

4.2 TRAP/J Operation

TRAP/J is an instance of TRAP in Java. To augment an existing Java program with the required hooks, TRAP/J uses the compile-time aspect weaving facilities provided in AspectJ. TRAP/J leverages Java structural reflection to support dynamic adaptation.

4.2.1 Overview

TRAP/J operates in two phases. The first phase takes place at compile time, when TRAP/J converts an existing Java program into an adapt-ready program. Figure 4.1 shows a high-level representation of TRAP/J operation at compile time. The application source code is compiled using the Java compiler (javac), and the compiled classes and a file containing a list of class names are input to an Aspect Generator and a Reflective Class Generator. For each class name in the list, these generators produce one aspect, one wrapper-level class, and one meta-level class. Next, the generated aspects and reflective classes, along with the original application source code, are passed to the AspectJ compiler (ajc), which weaves the generated and original source code together to produce an adapt-ready application. The second phase occurs at run time. New behavior can be introduced to the adapt-ready application using the wrapper- and meta-level classes (henceforth referred to as the adaptation infrastructure).

Figure 4.2 illustrates the interaction among the Java Virtual Machine (JVM) and the administrative consoles (GUI). First, the adapt-ready application is loaded by the JVM. At the time each metaobject is instantiated, it registers itself with the Java rmiregistry using a unique ID. Next, if an adaptation is required, the composer dynamically adds new code to the adapt-ready application at run time, using Java RMI to interact with the metaobjects. As part of the behavioral reflection provided in the adaptation infrastructure, a meta-object

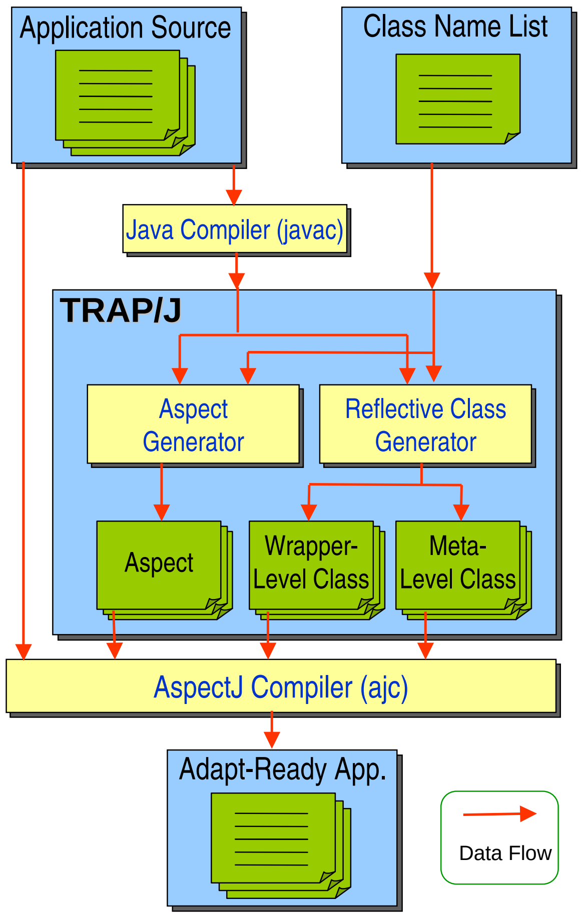


Figure 4.1: TRAP/J operation at compile time.

protocol (MOP) is supported in TRAP/J that allows interception and reification of method invocations targeted to objects of the classes selected at compile time to be adaptable.

4.2.2 TRAP/J Run-Time Model

To illustrate the operation of TRAP/J, let us consider a simple application comprising two classes, Service and Client, and three objects, (client, s1, and s2). Figure 4.3 depicts a simple run-time class graph for this application that is compliant with the run-time architecture of most class-based object-oriented languages. The class library contains Service and Client classes, and the heap contains client, s1, and s2 objects. The "instantiates" relationship among objects and their classes are shown using dashed arrows, and the "uses" relationships among objects are depicted with solid arrows.

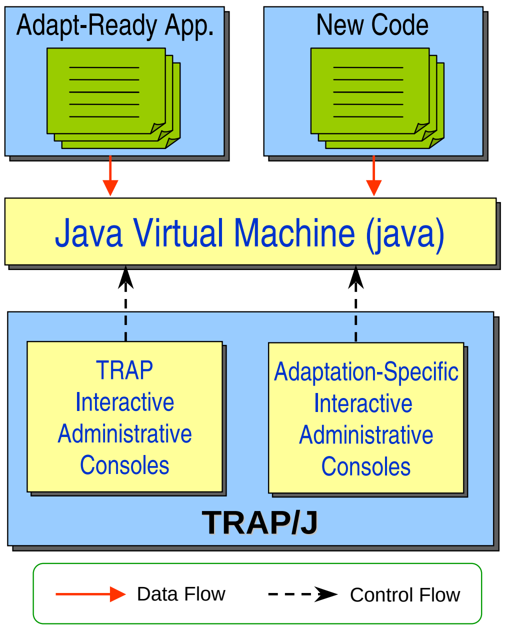

Adapt-Ready App.
New Code
Java Virtual Machine (java)
TRAP Interactive Administrative Consoles
Adaptation-Specific Interactive Administrative Consoles
TRAP/J
Data Flow
Control Flow

Figure 4.2: TRAP/J run-time support.

Figure 4.4 illustrates a layered run-time class graph model for this application. Please note that the base-level layer depicted in Figure 4.4 is equivalent to the class graph illustrated in Figure 4.3. For simplicity, only the "uses" relationships are represented in Figure 4.4. The wrapper level contains the generated wrapper classes for the selected subset of base-level classes and their corresponding instances. The base-level client objects use these wrapper-level instances instead of base-level service objects. As shown, s1 and s2 no longer refer to objects of the type Service, but instead refer to objects of type ServiceWrapper class. The meta level contains the generated meta-level classes corresponding to each selected base-level class and their corresponding instances. Each wrapper class has exactly one associated meta-level class, and associated with each wrapper object can be at most one metaobject. Please note that the behavior of each object in response to each message is dynamically programmable, using the generic method execution MOP provided in TRAP/J.

Finally, the delegate level contains adaptive code that can dynamically override base-level methods that are wrapped by the wrapper classes. Adaptive code is introduced into

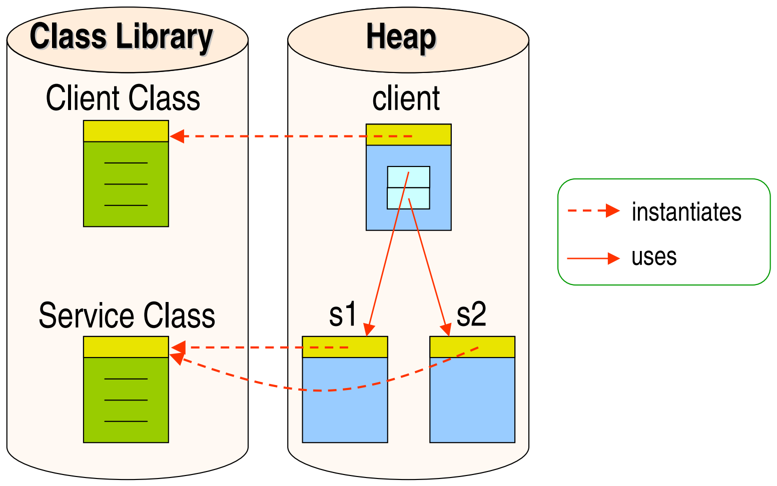


Figure 4.3: A simplified run-time class graph.

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

For example, let us assume that we want to adapt the behavior of a socket object (instantiated from a Java socket class such as the Java.net.MulticastSocket class) in an existing Java program at run time. First, at compile time, we use TRAP/J generators to generate the wrapper and metaobject classes associated with the socket class. Next, at run time, a composer can program the metaobject associated with the socket object to support dynamic reconfiguration. Programming the metaobject can be done by introducing a delegate class to the metaobject at run time. The metaobject then loads the delegate class, instantiates an object of the delegate class, intercepts all subsequent messages originally targeted to the socket object, and forwards the intercepted messages to the delegate object. Let us assume that the delegate object provides a new implementation for the `send(...)` method

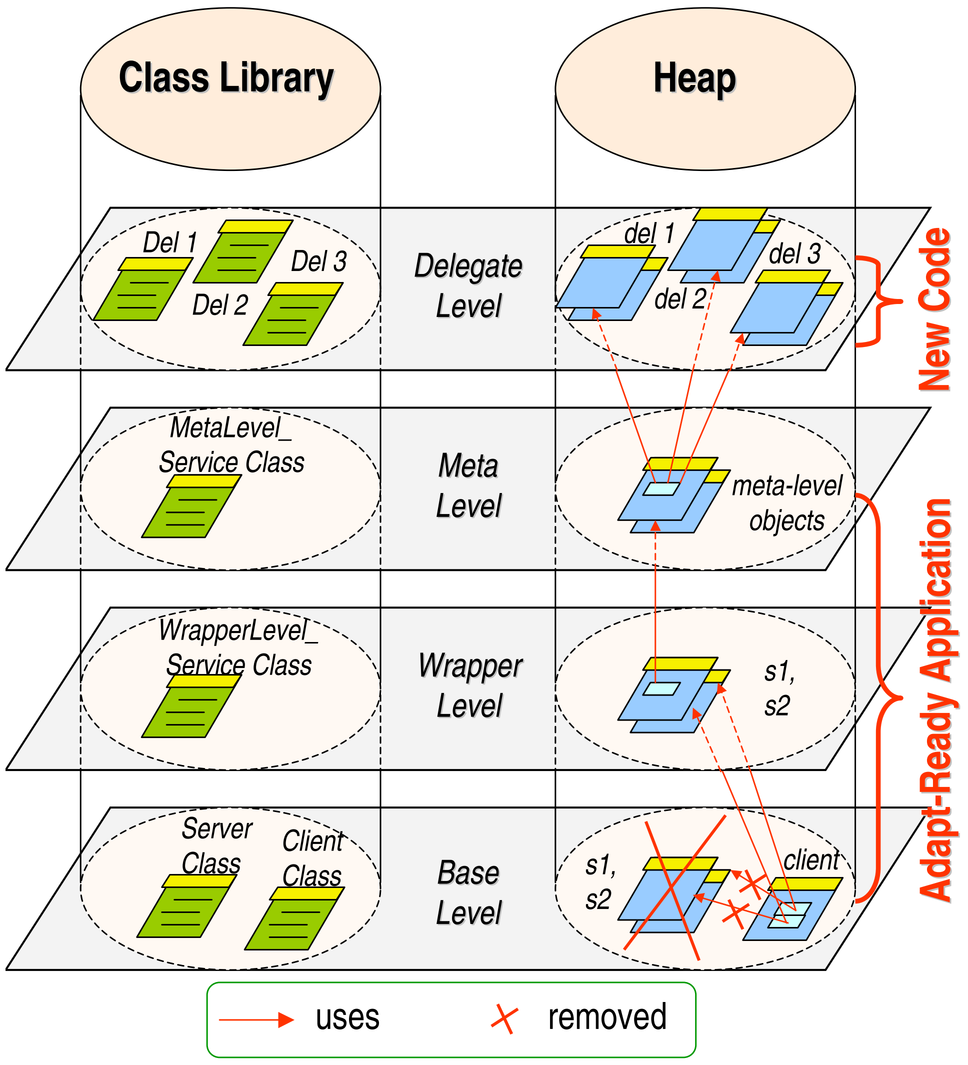


Figure 4.4: TRAP layered run-time model.

of the socket class. In this case, all subsequent messages to the `send(...)` method are handled by the delegate object and the other messages are handled by the original socket object. Alternatively, the delegate object could modify the intercepted messages and then forward them back to the socket object, resulting in a new behavior. TRAP/J allows the composer to remove delegates at runtime, bringing the object behavior back to its original implementation. Thus, TRAP/J is a non-invasive [155] approach to dynamic adaptation.

4.3 Case Study: Transparent Shaping of ASA

To demonstrate how TRAP/J can be used to produce adaptable programs from an existing program without the need to modify the existing program source code directly, we use the same example application used in the previous Chapter. For completeness, a brief description of ASA and our adaptation strategy is provided.

4.3.1 Example Application

ASA, introduced in Section 3.3.1, is an audio streaming application designed to stream interactive audio from a microphone at one network node to multiple receiving nodes. The original application was developed for wired networks. Our goal is to adapt this application to wireless environments, where the packet loss rate is dynamic and location dependent.

In this case study, we configured the experiments in an *ad hoc* wireless network as illustrated in Figure 4.5. A laptop workstation transmits an audio stream to multiple wireless iPAQs over an 802.11b (11Mbps) ad hoc wireless local area network (WLAN). Please note that unlike in wired networks, in wireless networks factors such as signal strength, interference, and antenna alignment produce dynamic and location-dependent packet losses. In current WLANs, these problems affect multicast connections more than unicast connections, since the 802.11b MAC layer does not provide link-level acknowledgements for multicast frames.

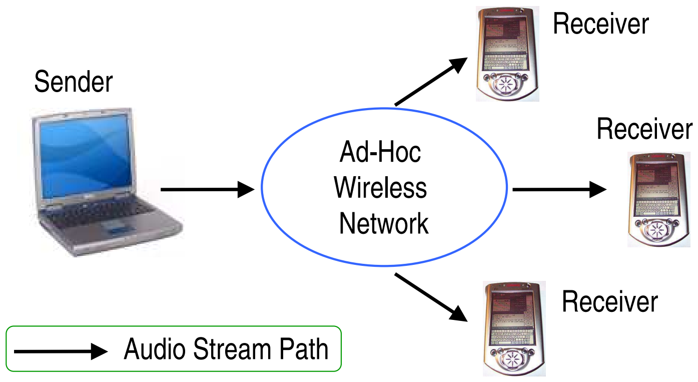


Figure 4.5: Audio streaming in a wireless LAN.

Figure 4.6 illustrates the strategy we used to enable ASA to adapt to variable channel conditions in wireless networks. This is the same strategy as the one used in Section 3.4. However, we used TRAP/J to modify ASA *transparently* such that it uses MetaSockets instead of Java multicast sockets. The particular MetaSocket adaptation used here is the dynamic insertion and removal of *forward-error correction* (FEC) filters [137]. Specifically, an FEC encoder filter can be inserted and removed dynamically at the sending MetaSocket, in synchronization with an FEC decoder being inserted and removed at each receiving MetaSocket. Use of FEC under high packet loss conditions reduces the packet loss rate as observed by the application. Under low packet loss conditions, however, FEC should be removed so as not to waste bandwidth on redundant data.

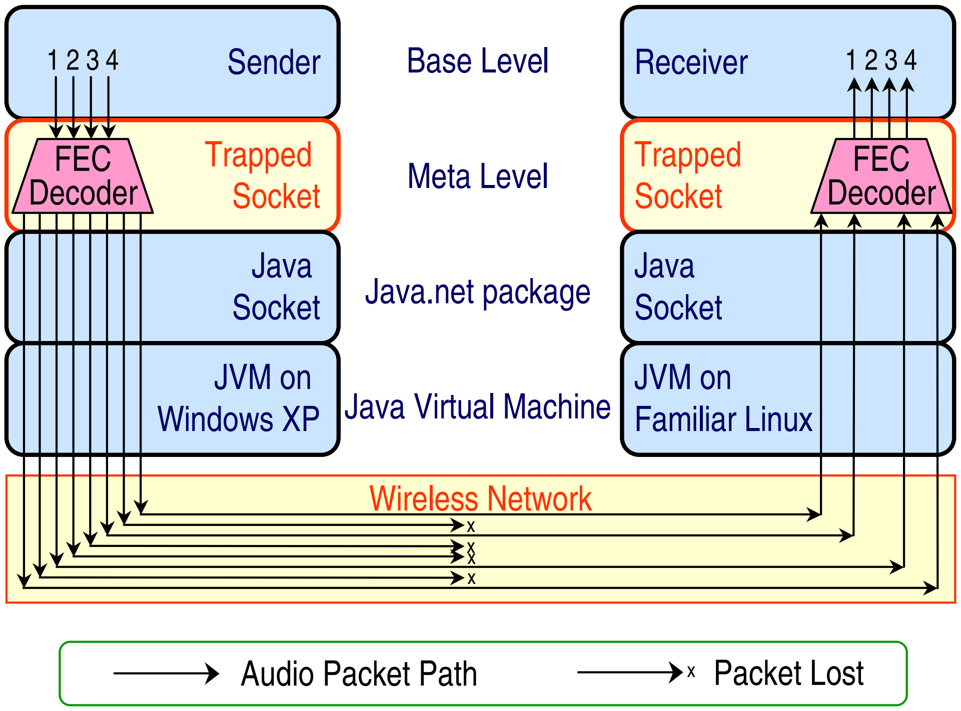


Figure 4.6: Adaptation strategy.

4.3.2 Making ASA Adapt-Ready

Figure 4.7 shows excerpted code for the original Sender class. The main method creates a new instance of the Sender class and calls its run method. The run method first creates an instance of AudioRecorder and MulticastSocket and assigns them to the instance variables, ar and ms, respectively. The multicast socket (ms) is used to send the audio datagram packets to the receiver applications. Next, the run method executes an infinite loop that, for each iteration, reads live audio data and transmits the data via the mulitcast socket.

```
public class Sender
{
  AudioRecorder ar;
  MulticastSocket ms;
  public void run()
  { . . .
    ar = new AudioRecorder(. . . );
    ms = new MulticastSocket();
    byte[] buf = new byte[500];
    DatagramPacket packetToSend =
      new DatagramPacket(buf, buf.length,
      target_address, target_port);
    while (!EndOfStream)
    {
      ar.read(buf, 0, 500);
      ms.send(packetToSend);
    } // end while . . .
  }
} // end Sender
```

Figure 4.7: Excerpted code for the Sender class.

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

Generated Aspect. The aspect generated by TRAP/J defines an initialization `pointcut` and the corresponding `around` advice for each `public` constructor of the MulticastSocket class. An `around` advice causes an instance of the generated wrapper class, instead of an instance of MulticastSocket, to serve the sender. Figure 4.8 shows excerpted code for the generated Absorbing_MulticastSocket aspect. This figure shows the "initialization" `pointcut` (lines 3-4) and its corresponding `advice` (lines 6-11) for the MulticastSocket constructor used in the Sender class. Referring back to the layered class graph in Figure 4.4, the sender (client) uses an instance of the wrapper class instead of the base class. In addition to handling `public` constructors, TRAP/J also defines a `pointcut` and an `around` advice to intercept all `public final` and `public static` methods.

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

Figure 4.8: Excerpted generated aspect code.

Generated Wrapper-Level Class. Figure 4.9 shows excerpted code for the WrapperLevel_MulticastSocket class, the generated wrapper class for the MulticastSocket. This wrapper class extends the MulticastSocket class. All the `public` constructors are overridden by passing the parameters to the super class (base-level class) (lines 4-6). Also, all the `public` instance methods are overridden (lines 8-29).

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

Figure 4.9: Excerpted generated wrapper code.

To better explain how the generated code works, we walk through the details of how the send method is overridden, as shown in Figure 4.9. The generated send method first checks if the metaObject variable, referring to the metaobject corresponding to this wrapper-level object, is null (lines 11-12). If so, then the base-level (super) method is called, as if the base-level method had been invoked directly by another object, such as an instance of sender. Otherwise, a message containing the context information is dynamically created using Java reflection and passed to the metaobject (metaObject) (lines 14-28). It might be the case that a metaobject may need to call one or more of the base-level methods. To support such cases, which we suspect might be very common, the wrapper-level class provides access to the base-level methods through the special wrapper-level methods whose names match the base-level method names, but with an "`Orig_`" prefix.

Generated Meta-Level Class. Figure 4.10 shows excerpted code for MetaLevel_MulticastSocket, the generated meta-level class for MulticastSocket. This class keeps an instance variable, delegates, which is of type Vector and refers to all the delegate objects associated with a metaobject that implements one or more of the base-level methods. To support dynamic adaptation of the `static` methods, a meta-level class provides the staticDelegates instance variable and its corresponding insertion and removal methods (not shown). *Delegate* classes introduce new code to applications at run time by overriding a collection of base-level methods selected from one or more of the *adaptable* base-level classes. An adaptable base-level class has corresponding wrapper- and meta-level classes, generated by TRAP/J at compile time. metaobjects can be programmed dynamically by inserting or removing delegate objects at run time. To enable a user to change the behavior of a metaobject dynamically, the meta-level class implements the DelegateManagement interface, which in turn extends the Java RMI Remote interface (lines 5-10). A composer can remotely "program" a metaobject through Java RMI. The insertDelegate and removeDelegate methods are developed for this purpose.

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

```
public class MetaLevel_MulticastSocket
  extends UnicastRemoteObject
  implements MetaLevel_Interface,DelegateManagement{

  private Vector delegates = new Vector();
  public synchronized void insertDelegate
    (int i, String delegateClassName)
    throws RemoteException { . . . }
  public synchronized void removeDelegate(int i)
    throws RemoteException { . . . }

  public synchronized Object invokeMetaMethod
    (Method method, Object[] args,
    ChangeableBoolean isReplyReady) throws Throwable{
    // Finding a delegate that implements this method
    . . .
    if(!delegateFound) // No meta-level method available
      throw new MetaMethodIsNotAvailable();
    else
      return newMethod.invoke(delegates.get(i-1),
        tempArgs);
}
```

Figure 4.10: Excerpted generated metaobject code.

4.3.3 Adapting to Loss Rate

To evaluate the TRAP/J-enhanced audio application, we conducted two sets of experiments similar to those in the previous chapter. The configuration use in these sets of experiments are illustrated in Figure 4.5.

In the first sets of experiments, a user holding a receiving iPAQ is walking within the wireless cell, receiving and playing a live audio stream. Figure 4.11 shows a sample of the results. For the first 120 seconds, the program has no FEC capability. At 120 seconds, the user walks away from the sender and enters an area with loss rate around 30%. The adaptable application detects the high loss rate and inserts a (4,2) FEC filter, which greatly reduces the packet loss rate as observed by the application, and improves the quality of the audio as heard by the user. At 240 seconds, the user approaches the sender, where the network loss rate is again low. The adaptable application detects the improved transmission and removes the FEC filters, avoiding the waste of bandwidth with redundant packets.

Again at 360 seconds, the user walks away from the sender, resulting in the insertion of FEC filters. This experiment demonstrates the utility of TRAP/J to transparently and automatically enhance an existing application with new adaptive behavior.

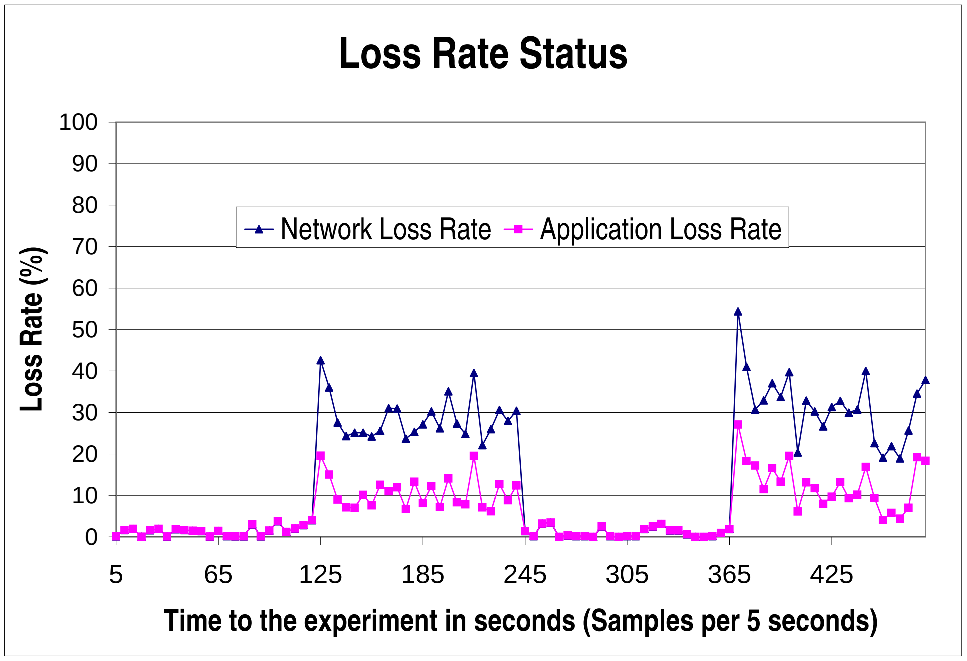


Figure 4.11: The effect of using FEC filters to adapt ASA to high loss rates on a wireless network.

4.3.4 Balancing QoS and Energy Consumption

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

Figure 4.12 shows a trace of an experiment using the ASA described earlier, running in ad hoc mode. A stationary user speaks into a laptop microphone, while another user listens on an iPAQ as he changes his location in the wireless cell from time to time. In this particular test, the iPAQ user remains in a low packet loss area for approximately 30 minutes, moves to a high packet loss area for another 40 minutes, moves back to the low packet loss location for another 30 minutes, then reenters the high packet loss location. He remains there until the iPAQ's external battery drains and the network is disconnected.

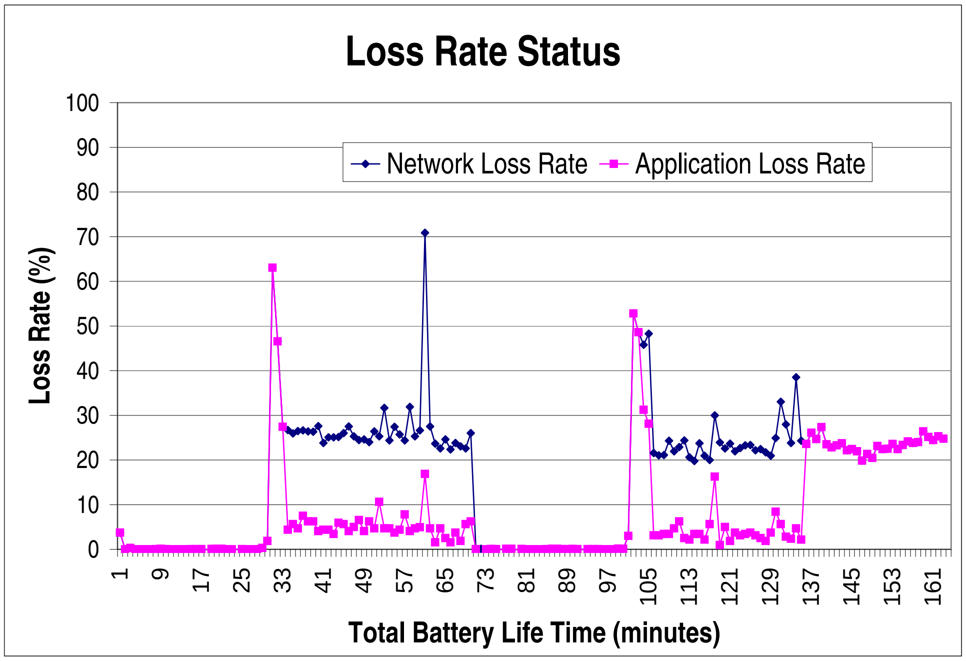


Figure 4.12: MetaSocket packet loss behavior with dynamic FEC filter insertion and removal.

In this experiment, the upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 20%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 5%. As shown in Figure 4.12, the FEC $(4, 2)$ code is effective in reducing the packet loss rate as observed by the application. Figure 4.13

plots the remaining battery capacity as measured during the above experiment and that for a non-adaptive trace. The adaptive version extends the battery lifetime by approximately 27 minutes.

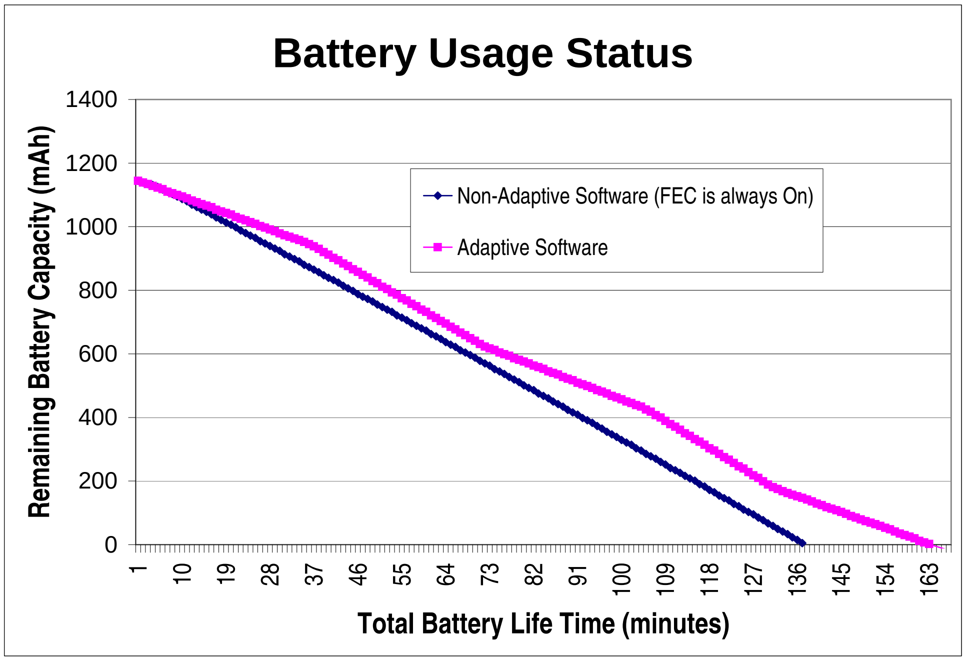


Figure 4.13: Trace of energy consumption during experiment using a software measurement technique.

4.4 Related Work

Like TRAP/J, many approaches to constructing adaptable programs involve *intercepting* interactions among objects in functional code, and *redirecting* them to adaptive code. We identify three categories of related work.

The first category includes approaches that *extend middleware* to support adaptive behavior. Since the traditional role of middleware is to hide resource distribution and platform heterogeneity from the business logic of applications, it is a logical place to put adaptive behavior related to other cross-cutting concerns, such as quality-of-service, energy management, fault tolerance, and security [6, 37, 44, 45, 92, 99, 113, 114, 116, 117, 119, 121, 122, 127, 156–158]. In addition to providing transparency to the functional code, some approaches provide transparency to the distribution middleware code as well. For example, IRL [116] and FTS [156, 159] use CORBA portable interceptors [47] to intercept CORBA messages transparently, and Eternal [121] intercepts calls to the TCP layer using the Linux /proc file system. Adaptive middleware approaches provide an effective means to support adaptability, but they are applicable only to programs that are written for a specific middleware platform such as CORBA, Java RMI, or DCOM/.NET. We discuss approaches in this category in more details in the next chapter.

The second category provides such transparency by *extending Java virtual machines* with facilities to intercept and redirect interactions in the functional code [1]. Examples of extensions to Java virtual machines (JVMs) include PROSE [32], Iguana/J [98], metaXa [153], Guaraná [140], and R-Java [141]. These projects employ a variety of techniques. For example, Guaraná extends the Kaffe open source JVM [160], whereas PROSE and Iguana/J extend the standard JVM using command-line parameters (*e.g.*, `-Xbootclasspath` and various HotSpot options) to introduce their specific JIT compilers to the JVM and to disable the Java HotSpot option. Guaraná and Iguana/J employ meta-object protocols to provide dynamic adaptation to existing Java programs, whereas PROSE employs a dynamic aspect weaving technique for this purpose, without modifying the program and JVM source code. In general, approaches in this category are very flexible with respect to dynamic reconfiguration, in that new code can be introduced to the application at run time. Iguana/J supports *unanticipated* adaptation at run time by allowing new MOPs to be associated with classes and objects of a running application, without the need for any pre- or post-processing of the application code at compile or load time. However, while these solutions provide transparency with respect to the application source code, extensions to the JVM may reduce their portability.

Finally, the third category includes approaches that transparently *augment the applica-*

[1]According to the taxonomy of adaptive middleware introduced in Chapter 2, we consider JVM as host-infrastructure middleware. However, for their specific characteristics, here we consider extensions to JVM in a separate category than the middleware extensions category.

tion code itself with facilities for interception and redirection. Example projects include OpenJava [148], FRIENDS [94], PCL [97], AspectJ [103], Composition Filters [78], ARCAD [27], Hyper/J [104], DemeterJ (DJ) [105], JAC [106], Reflex [151], Kava [107], Dalang [161], Javassist [147], and JOIE [48]. Most of these systems are designed to work in two phases. In the first phase, interception hooks are woven into the application code at either compile time, using a pre- or post-processor, or at load time, using a specialized class loader. For example, AspectJ enables aspect weaving at compile time. In contrast, Reflex and Kava use bytecode rewriting at load time to support transparent generation of adaptable programs. In the second phase, intercepted operations are forwarded to adaptive code using reflection.

TRAP/J belongs to this last category and employs a two-phase approach to adaptation. TRAP/J is completely transparent with respect to the original application source code and does not require an extended JVM. By supporting compile-time selection of classes for possible later adaptation, TRAP/J enables the developer to balance flexibility and efficiency. Reflex [151] also address this issue by allowing a meta-level architect to select an application-specific MOP that best fits the application requirements. A default MOP can be used when an application-specific MOP is not needed. TRAP/J complements Reflex by providing a generic MOP that could serve as the default MOP. TRAP/J is most similar to ARCAD [27], which also provides a two-phase approach to dynamic adaptation. ARCAD also uses AspectJ at compile time and behavioral reflection at run time. However, the partial behavioral reflection [151] provided in TRAP/J is more fine-grained than that of ARCAD. Specifically, TRAP/J supports method invocation reflection, enabling an arbitrary subset of an object's methods to be selected for interception and reification; ARCAD does not support such fine-grained reflection. The ability of TRAP/J to avoid unnecessary reifications is due to its multi-layer architecture.

4.5 Summary

In this chapter, we introduced TRAP, which is a language-based approach to transparent shaping. TRAP enables production of adaptable program families from existing programs developed in class-based, object-oriented programming languages. We described the design and operation of TRAP/J, which is an instance of TRAP in Java. TRAP/J enables dynamic reconfiguration of Java applications without the need to modify the application source code directly and without extending the JVM. TRAP/J operates in two phases. At compile time, TRAP/J produces an adapt-ready version of an existing Java program. Later at run time, TRAP/J enables adding new behavior to the adapt-ready program dynamically through insertion and removal of delegates into the adapt-ready program. As such, other members of the adaptable program family associated with the adapt-ready program can be produced dynamically. A case study in a wireless network environment was used to demonstrate the operation and effectiveness of TRAP/J.

In TRAP/J, an adaptation hook is realized by a pair of wrapper and meta classes associated with a class in an existing Java program, and adaptive code is realized by delegates, which can modify the behavior of the class by overriding the implementation of its methods. We developed a delegate using a MetaSocket, which in its turn supports dynamic adaptation through insertion and removal of filters. As a result, unlike the approach in the previous chapter, at run time, a MetaSocket can be replaced with a more appropriate one, if required.

Although TRAP/J is not an adaptive middleware, it can be used to weave adaptive middleware components (*e.g.,* MetaSockets) into distributed applications. Therefore, we classify it according to the taxonomy introduced in Section 2. First, TRAP/J operates in the application layer because it is a language-based approach that can be used to transform existing application code. Second, TRAP/J can be used to transparently weave adaptive middleware services into applications, so it supports an intercepting technique for accessing middleware services. Finally, TRAP/J supports tunable adaptation since the original application code remains intact during run time.

As part of our future studies, we plan to develop TRAP/C++ and TRAP/C# to provide dynamic adaptation in existing C++ and C# programs, respectively.

Chapter 5

Transparent Shaping of CORBA Programs

Although TRAP is transparent to application code, the adaptation behavior is still woven into the application, which must be recompiled. Implementing transparent shaping in middleware can produce even greater transparency. As observed by several researchers [6, 28, 37–45, 91, 92, 99, 113, 114, 116, 117, 119, 121, 122, 127, 156–158, 162], middleware is a natural place to incorporate adaptive behavior and to hide unanticipated conditions from existing distributed applications. In this part of our study, we investigate enhancements to CORBA ORBs to support dynamic reconfiguration of middleware services transparently both to the application and middleware code. Moreover, we address interoperation among otherwise incompatible adaptive middleware frameworks (*e.g.*, QuO [42] and Open ORB [38]) to enable existing programs to benefit from more than one adaptive framework. The result of this study is a middleware-based extension of transparent shaping, which supports dynamic adaptation in existing CORBA applications. We call this programming model the *Adaptive CORBA Template (ACT)*.

As an extension of transparent shaping, ACT can be used to produce an adapt-ready version of an existing CORBA program by introducing a hook, which intercepts all CORBA remote interactions, into the program at compile time. To do so, ACT uses CORBA portable interceptors [47], supported in CORBA compliant ORBs (described later). Portable interceptors can be incorporated into a CORBA program at startup time using a command-line parameter. Later at run time, these hooks can be used to insert adaptive code into the adapt-ready program, which in turn can adapt the requests, replies, and exceptions passing through the ORBs. In this manner, ACT enables run-time improvements to the program in response to unanticipated changes in its execution environment, effectively producing other members of the adaptable program family dynamically.

We refer to ACT as a *template*, because it is independent of programming languages and CORBA ORB implementations. As depicted in Figure 5.1, ACT can be instantiated in a programming language, such as Java and C++, that supports dynamic code loading and is supported by CORBA. Moreover, ACT can be used to extend existing adaptive CORBA frameworks such as QuO [42]. To evaluate the performance and functionality of ACT, we constructed a prototype of ACT in Java, called *ACT/J*. ACT/J supports unanticipated adaptation for crosscutting concerns such as QoS and system-resource management. Our experimental results show that the overhead introduced by the ACT/J infrastructure is negligible, while the adaptations offered are highly flexible.

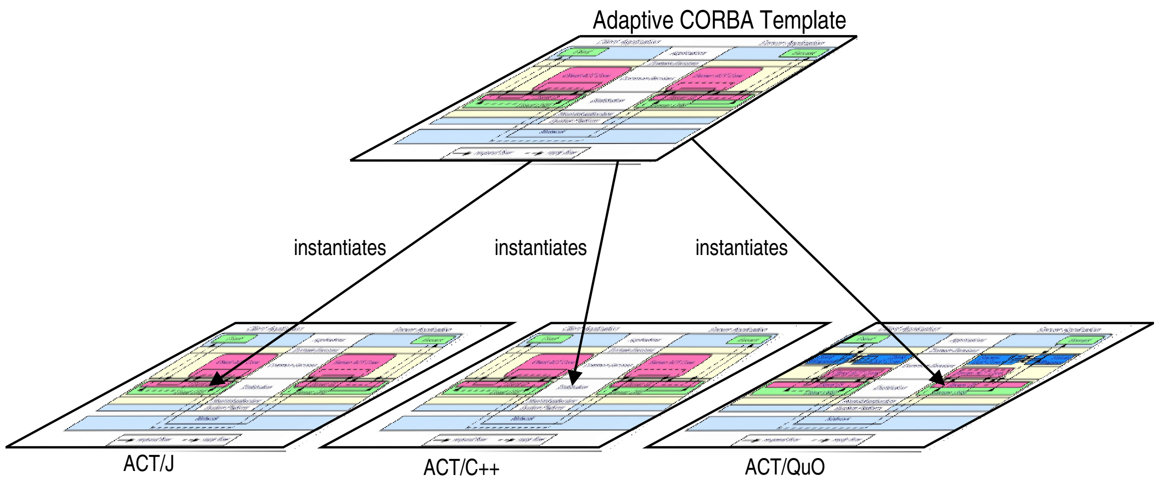


Figure 5.1: ACT as a template that can be instantiated in different programming languages and can be used to enhance existing adaptive CORBA frameworks.

The remainder of this chapter is organized as follows. Section 5.1 provides a background on CORBA portable interceptors. Section 5.2 describes the ACT architecture. Section 5.3 introduces the generic proxy, which facilitates transparent development of adaptive code with respect to application code. Section 5.4 introduces ACT/J and its operation. Section 5.5 describes a case study, where we used the generic proxy in ACT/J to implement transparent self-optimization in an existing CORBA application, enabling it to accommodate changing conditions of a wireless network infrastructure. Section 5.6 describes another case study, where we coupled ACT and QuO, called *ACT/QuO*, as an example of how ACT enables integration of different middleware frameworks. Section 5.7 categorizes research projects and discusses how ACT relates to other approaches. Finally, Section 5.8 summarizes the chapter.

5.1 Background

In this section, we provide a background on CORBA and review CORBA portable interceptors as defined by OMG [47]. Although we have briefly introduce CORBA in Section 2, here we introduce it again with more details for completeness.

5.1.1 CORBA

The *Common Object Request Broker Architecture (CORBA)* [47] is a distributed object framework proposed by the Object Management Group (OMG). CORBA supports distributed object-oriented computing across heterogeneous hardware devices, operating systems, network protocols, and programming languages. Figure 5.2 illustrates the CORBA components described as follows. 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 of a CORBA object first acquires a reference to the CORBA object using either an interoperable object reference (IOR) file or a CORBA naming service [47]. Next, the client calls methods on this reference as if the 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.

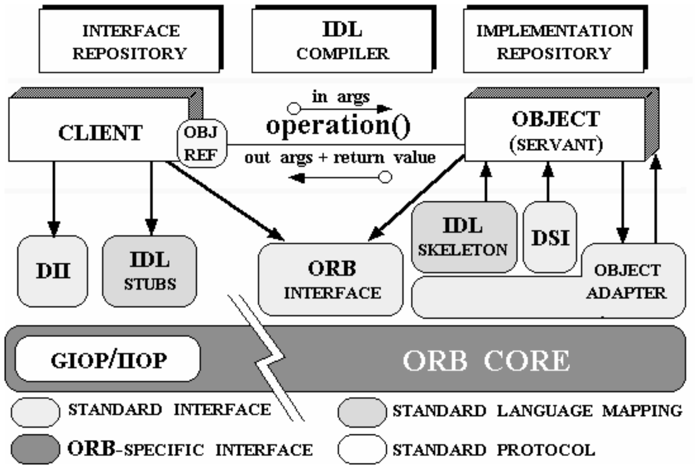


Figure 5.2: CORBA architecture [10].

The *dynamic invocation interface (DII)* enables clients to directly access the underlying request mechanisms at run time to generate dynamic requests to objects, whose type (interface) were not known at the time the client program was compiled. The *interface repository* provides the type information that a client needs to dynamically create a request. The *dynamic skeleton interface (DSI)* enables an ORB to deliver requests to a servant, which does not have compile-time knowledge of the type of the object it supports (*e.g.*, a gateway object may not know the type of the target objects to which it is forwarding requests). The *implementation repository* enables late deployment of CORBA objects. The implementation repository receives the first request targeted to a CORBA object, looks up the object meta information in its database, activates the object, and forwards the request "permanently" to the target object. Permanent forwarding, in contrast to transient forwarding, also causes automatic forwarding of all future requests from the same client and to the same target object directly from the client ORB. The *object adapter* activates servants and dispatches requests to them. The *ORB interface* provides access to standard ORB services, such as resolving the CORBA initial services such as the CORBA naming service. The *general inter-ORB protocol (GIOP)* is a standard for inter-ORB communication, which enables interoperability among different CORBA-compliant ORBs. The *Internet inter-ORB protocol (IIOP)* is a specific mapping of the GIOP specification developed to use the TCP/IP protocol stack.

5.1.2 CORBA Portable Request Interceptors

CORBA Portable Request Interceptors provide a transparent mechanism to intercept messages (reified requests, replies, and exceptions) inside the ORBs of a CORBA application. According to the specification, a request interceptor is considered as part of an ORB and must be registered with the ORB at its initialization time (notably, a request interceptor cannot be registered with the ORB at run time). Figure 5.3 shows the flow of a CORBA request/reply sequence with interceptors present in a typical CORBA application. This application comprises two autonomous programs hosted on two computers connected by a network. Let us assume that the client has a valid CORBA reference to the CORBA object realized by the servant. The client's request to the servant is first received by the stub, which represents the CORBA object at the client side. The stub marshals the request and sends it to the client ORB, where the request is intercepted by the client request interceptor. The interceptor can inspect requests, create new requests, and raise exceptions. For example, the ForwardRequest exception can be used to forward a particular request to a *different* CORBA object. However, to ensure portability, interceptors are not allowed to reply to intercepted requests or to modify the parameters [47]. This restriction limits the ability of request interceptors alone to adapt the behavior of CORBA applications.

Continuing the example, let us assume that the client-request interceptor in Figure 5.3 simply passes the request unmodified. In this case the client ORB sends the request to the server ORB, where it is intercepted by the server-request interceptor. Again, let us assume that the request is passed unmodified, in which case it is delivered to the servant by way of a skeleton, which unmarshals the request. The servant replies to the request, by way of

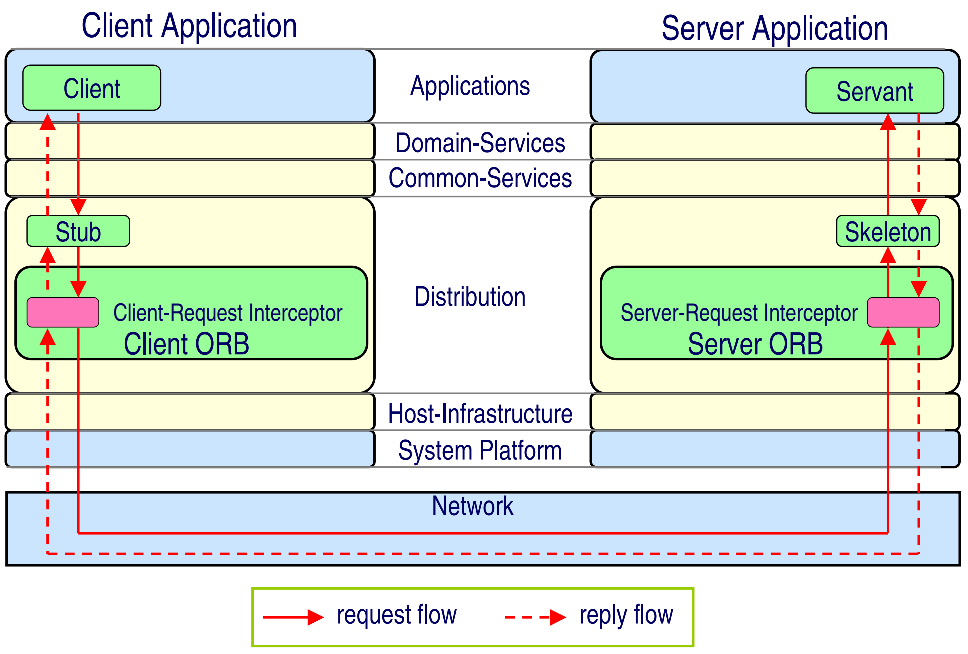


Figure 5.3: A simple CORBA application with request interceptors.

the server ORB, where the reply also is intercepted. Eventually, the reply will be received by the client ORB and is intercepted by the client-request interceptor before it reaches the client.

As we shall discuss in Section 5, the generic interceptors in ACT are in fact CORBA portable interceptors. The interceptors provide "hooks" into the interaction between clients and servants. Moreover, they use the ForwardRequest exception to deliver requests to a *proxy*, a CORBA object that is not prohibited from replying to or modifying the request.

5.2 ACT Architecture and Operation

ACT is intended to support the construction and enhancement of adaptive CORBA frameworks. ACT enables CORBA applications to support unanticipated adaptation at run time without the need to modify, recompile, and relink the application source code. We introduce ACT by defining its core components and by describing their interaction with the rest of the system.

5.2.1 ACT Core Components

Figure 5.4 shows the flow of a request/reply sequence in a simple CORBA application using ACT. 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 interceptor intercepts all the incoming requests and outgoing replies (or exceptions) and forwards them to its ACT core. A CORBA application is called *adapt-ready* if a generic interceptor is registered with all its ORBs at startup time. If, in addition to the generic interceptors, all the ACT core components are also loaded into the application, the application is called *ACT-ready*. Making the application ACT-ready can be done either at startup time or at run time.

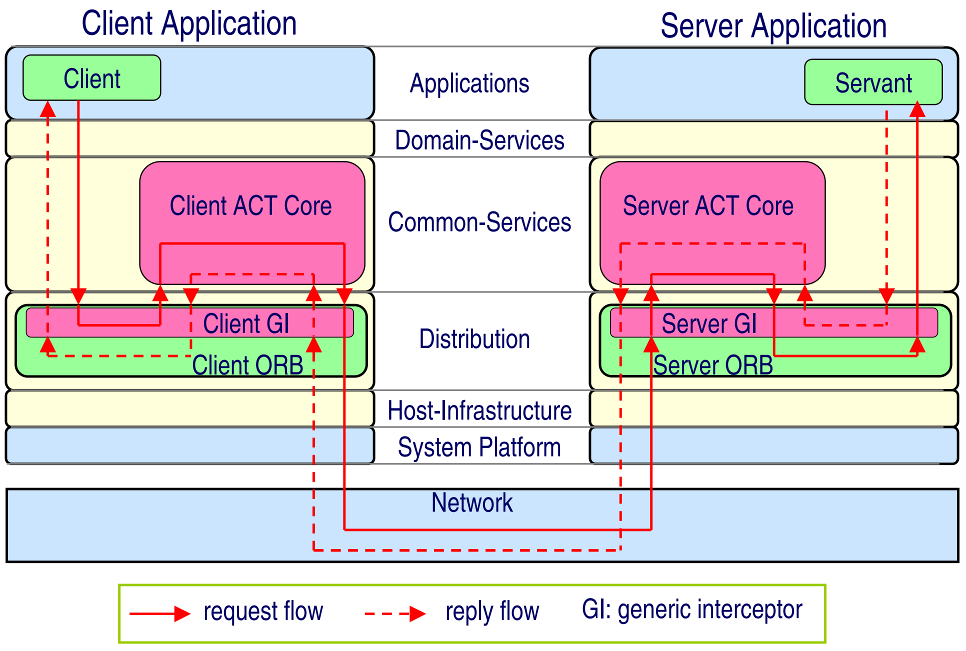


Figure 5.4: ACT configuration in the context of a simple CORBA application.

Figure 5.5 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.

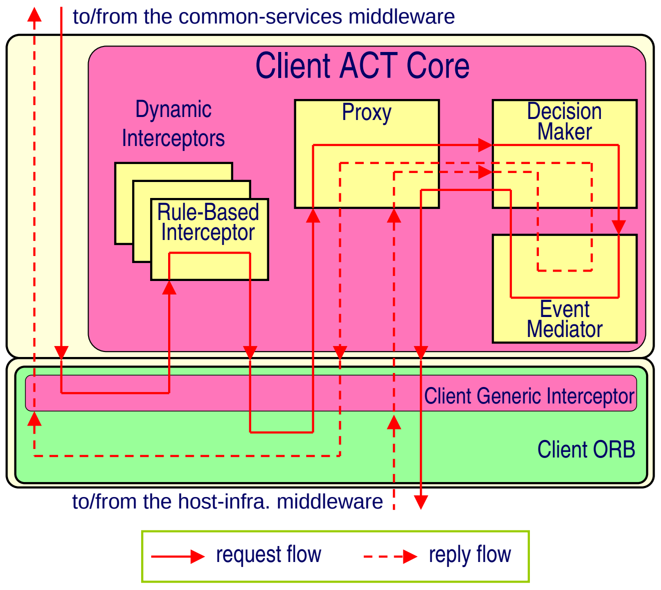


Figure 5.5: ACT core components interacting with the rest of the system.

Dynamic Interceptors. According to the CORBA specification [47], a request interceptor is required to be registered with an ORB at the ORB initialization time. The ACT core enables registration of request interceptors after the ORB initialization time (at run time) by publishing a CORBA interceptor-registration service. Such request interceptors are called *dynamic interceptors*. Dynamic interceptors can be unregistered with the ORB at run time also. In contrast, a request interceptor that is registered with the ORB at startup time is called a *static interceptor* and cannot be unregistered with the ORB during run time. We note that the code developed for a static interceptor and that for a dynamic interceptor can be identical, the difference being the time at which they are registered. In ACT, only generic interceptors are static.

A *rule-based interceptor* is a particular type of dynamic interceptor that uses a set of rules to direct the operations on intercepted requests. The rules can be inserted, removed, and modified at run time. A *rule* consists of two objects: a condition and an action. To determine whether a rule matches a request, a rule-based interceptor consults its condition object. Once a match is found, the interceptor sends the request to the action object of the rule. Since it is part of a CORBA portable interceptor, the action object cannot itself reply to the request or modify the request parameters [47]. The action object can, however, send new requests, record statistics, or raise a ForwardRequest exception, causing the request to be forwarded to another CORBA object such as a proxy.

Proxies. A *proxy* is a surrogate for a CORBA object that provides the same set of methods as the CORBA object. Unlike a request interceptor, a proxy is not prohibited from replying to intercepted requests. A proxy can reply to the intercepted request by sending a new request (possibly with modified arguments) to either the target object or to another object. Alternatively, a proxy can reply to the intercepted requests using local data (*e.g.,* cached replies).

Decision Makers. A *decision maker* assists proxies in replying to intercepted requests as depicted in Figure 5.5. 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.

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. [136]. 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.

5.2.2 Interaction among ACT Components

To describe the interactions among the ACT components, we provide a detailed sequence diagram [163] in Figure 5.6. The diagram shows the flow of a request/reply sequence in an ACT-ready application. The configuration shown in Figures 5.4 and 5.5 is used as the basis for this particular sequence diagram. Here, we consider only the activities on the client side and, for clarity, stubs and skeletons are not shown.

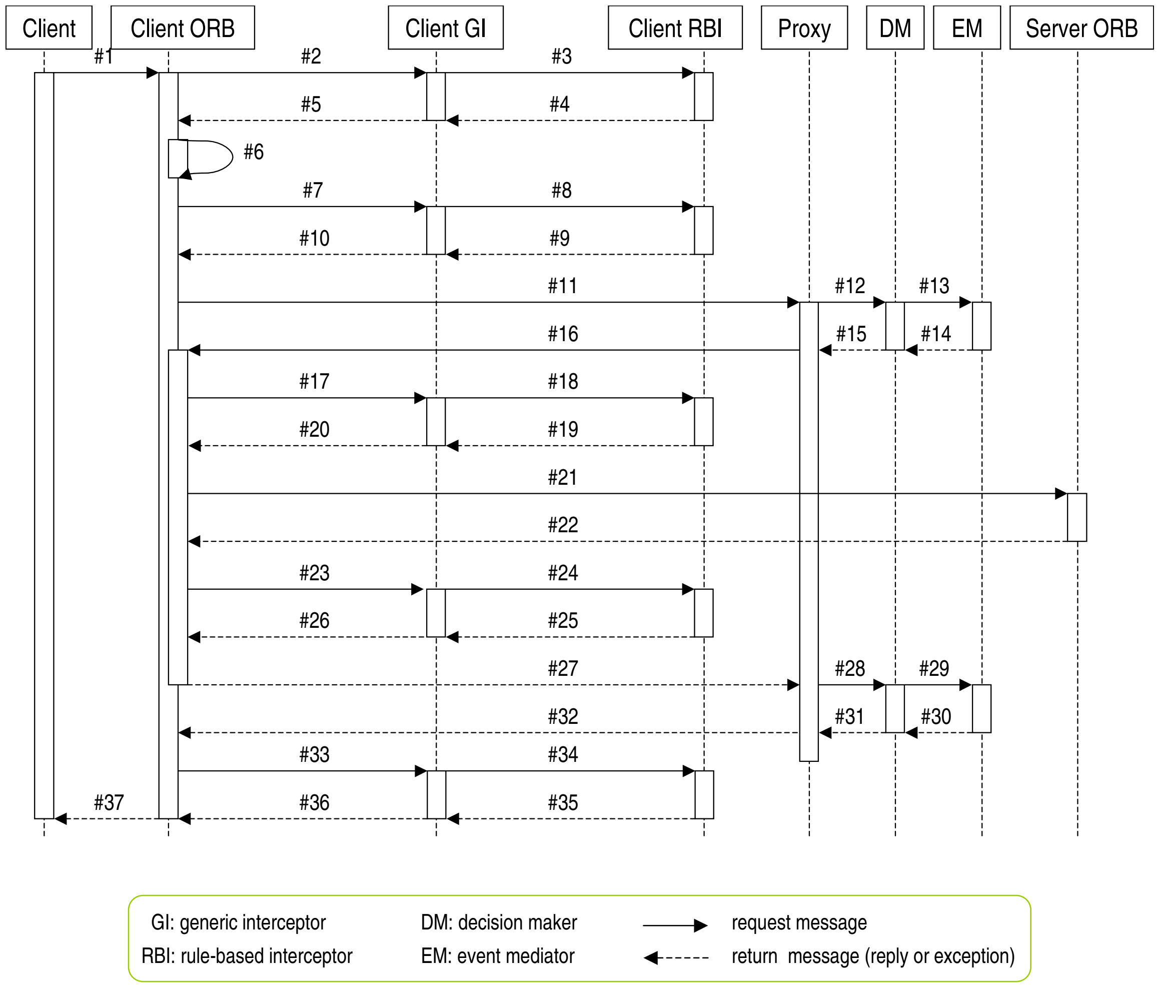


Figure 5.6: Request/reply sequence in the client side of an ACT-ready application.

First, the request from the client to the servant is forwarded to the proxy (messages #1 to #11). After the request is received by the client ORB (#1), it is intercepted by the client generic interceptor (#2), where it is forwarded to the client rule-based interceptor

(#3). The client rule-based interceptor checks its active rules. In this scenario, we assume it finds a rule that matches the request. The rule raises a ForwardRequest exception, which is passed to the client generic interceptor (#4) and then to the client ORB (#5), where the request target is changed to the proxy (#6). Before the new request is sent to the proxy, it is intercepted again by the client generic and rule-based interceptors (#7 and #8), but this time no exception is raised (#9 and #10), and the calls simply return. The proxy receives the request (#11).

Next, the proxy processes the request and forwards it to the servant (messages #12 to #21). The proxy consults the decision maker (#12), where an event may be raised to handle an unknown situation (#13 and #14). 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 (#15) via the client ORB (#16). The modified request is also intercepted by the client generic and rule-based interceptors (#17 and #18) but again no exception is raised (#19 and #20). Therefore, the modified request is sent to the server ORB (#21).

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 (#22). The reply is intercepted by the client generic and rule-based interceptors (#23 and #24), where no exception is raised (#25 and #26). The reply is sent back to the proxy (#27), where it is forwarded to the decision maker (#28) for possible modifications and possible event raising (#29, #30, and #31).

Finally, using the reply from the servant and the direction given by the decision maker, the proxy replies to the client's request (#32). The reply is intercepted by the client generic and rule-based interceptors (#33 and #34). Again no exception is raised (#35 and #36), and the client ORB sends the reply back to the client (#37).

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 5.6, indicate that the overhead is actually quite small.

5.3 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 defined in the application) of specific applications.

5.3.1 Architecture

The *generic proxy* is a particular CORBA object that is able to receive *any* CORBA request (hence the label "generic"). To determine how to handle a particular request, the generic proxy accesses the CORBA interface repository [47], 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 in this manner implies no need to modify or recompile the application source code. The interface repository, however, requires access to the CORBA IDL files used in the application.

In default operation, 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*.

5.3.2 Operation

Figure 5.7 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 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) [47] to receive any type of request. The generic proxy and the rule-based decision maker use the CORBA dynamic invocation interface (DII) [47] to create and invoke a new request dynamically.

5.4 ACT/J Implementation

We have developed an instance of ACT in Java, called *ACT/J*, to evaluate ACT in practice. ACT/J was tested over ORBacus [119], a CORBA-compliant ORB distributed by IONA Technologies. ORBacus [119], like JacORB [120], TAO [44], and many other CORBA ORBs, supports CORBA portable interceptors [47], the only requirement for using ACT.

To make a CORBA application ACT-ready at the application startup time, we need to resolve the following bootstrapping issues. First, we need to register a generic interceptor

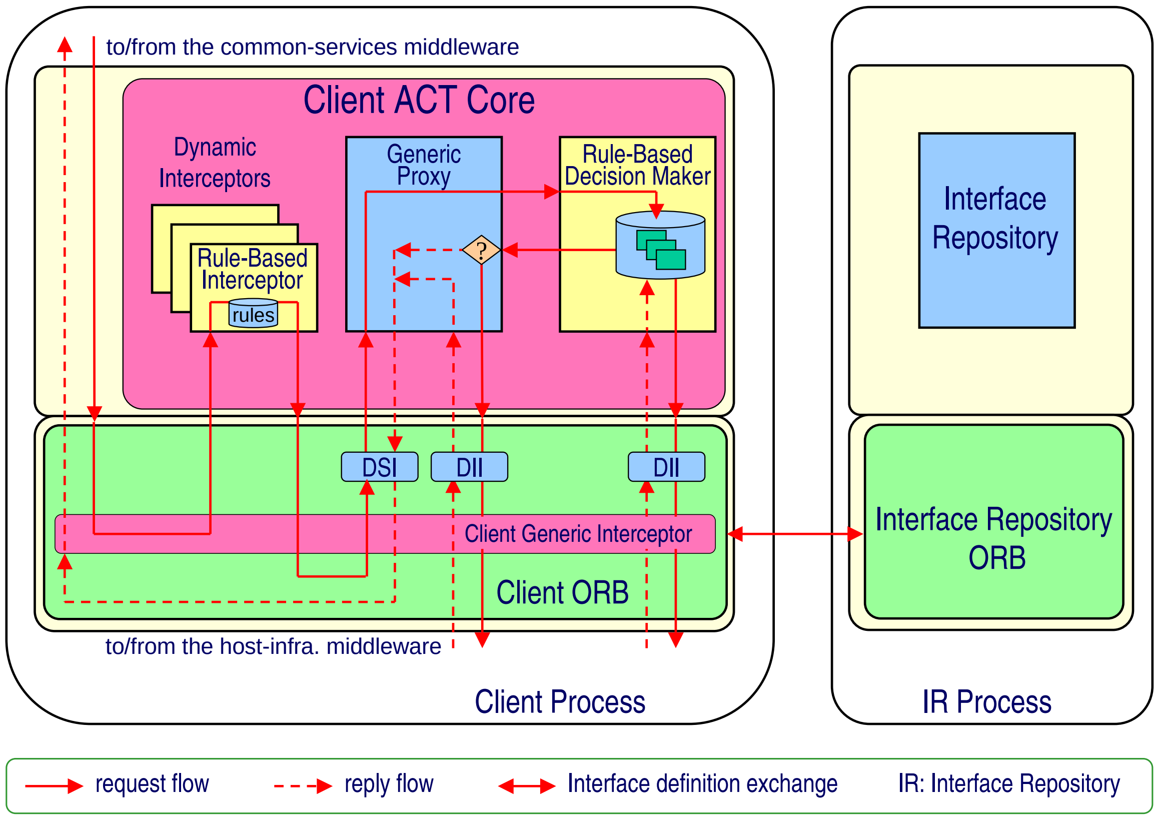
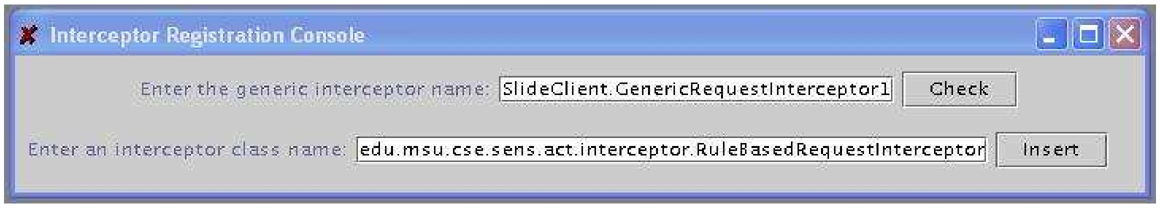


Figure 5.7: Incorporating generic proxy in the ACT core.

with the application ORB. Like many other ORBs, ORBacus [119] uses a configuration file that enables an administrator to register a CORBA portable interceptor with the application ORB. JacORB [120] and TAO [44] use a similar approach. Second, since the components in the ACT core are also CORBA objects, they require an ORB to support their operation (registration of services, and so on). Therefore, we need either to obtain a reference to the application ORB for this purpose, or to create a new ORB. ORBacus does provide such a reference, although the CORBA specification does not support this feature. To implement ACT/J over an ORB that does not provide such a reference, we simply create a new ORB, although its use introduces additional overhead.

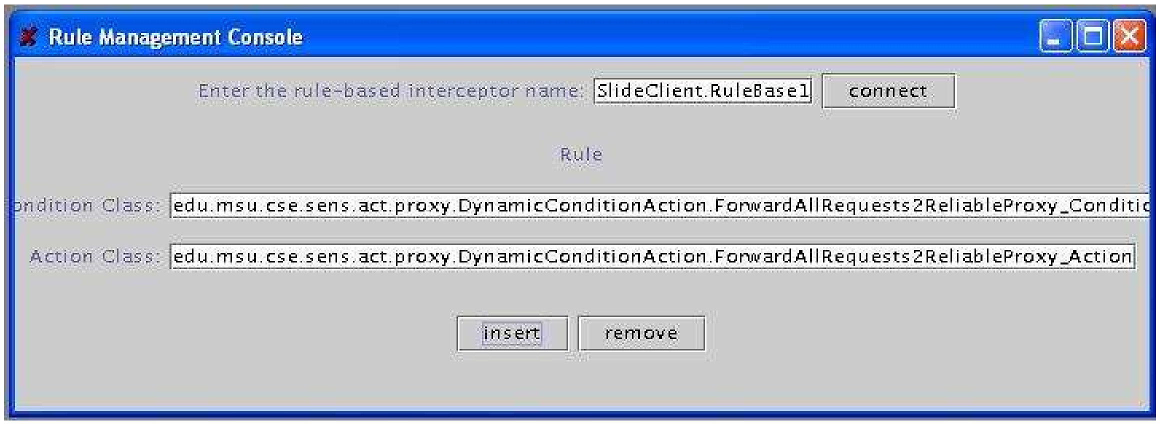
To test the operation of ACT/J, we developed two administrative consoles: the Interceptor Registration Console and the Rule Management Console. Please note that in this study the composer is assumed to be a human, who performs dynamic adaptation using the administrative consoles. Figure 5.8 shows the *Interceptor Registration Console*, which 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. Figure 5.9 shows the *Rule Management Console*, which allows a user to manually insert rules into rule-based interceptors.


Interceptor Registration Console

Enter the generic interceptor name: SlideClient.GenericRequestInterceptor1

Enter an interceptor class name: edu.msu.cse.sens.act.interceptor.RuleBasedRequestInterceptor

Figure 5.8: Interceptor Registration Console


Rule Management Console

Enter the rule-based interceptor name: SlideClient.RuleBase1

Rule

ondition Class: edu.msu.cse.sens.act.proxy.DynamicConditionAction.ForwardAllRequests2ReliableProxy_Conditic

Action Class: edu.msu.cse.sens.act.proxy.DynamicConditionAction.ForwardAllRequests2ReliableProxy_Action

Figure 5.9: Rule Management Console

5.5 Case Study: Transparent Self Optimization

To evaluate the effectiveness of ACT/J to support self-management in existing CORBA applications, without modifying the application code, we conducted a case study in which self-optimization is enabled in an existing application. Additional experiments involving IP handoff, are described in an accompanying technical report [164]. We begin with a brief overview of the application and the experimental environment, followed by the description of the experiment. The experiment shows how ACT/J could be used to support autonomic computing in either a generic or application-specific manner.

5.5.1 The Example Application and Experimental Environment

For the application, we adopted an existing distributed image retrieval application developed by BBN Technologies [165]. The application has two parts, a client that requests and displays images, and a server that stores the images and replies to requests for them. In this study, we treat the application as though it were used for surveillance, with a mobile user executing the client code on a laptop and monitoring a physical facility through continuous still images from multiple camera sources. For the experiment described later in this section, we executed the server on a desktop computer connected to a 100 Mbps wired network and the client on a laptop computer connected to a three-cell 802.11b wireless network. Both the desktop and laptop systems are running the Linux operating system.

Figure 5.10 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 11Mbps connections, whereas AP-2 provides only 2Mbps. The desktop running the server application is close to AP-1. AP-1 and AP-2 are managed by our Computer Science and Engineering Department, whereas AP-3 is managed by the College of Engineering. This difference implies that the IP address assigned to the client laptop needs to change as the user moves from a CSE wireless cell to a College cell.

Figure 5.11 shows an example image from the experiment. The server provides four different versions of each image, varying in size and quality. Typical comparative file sizes are 90KB, 25KB, 14KB, and 4KB.

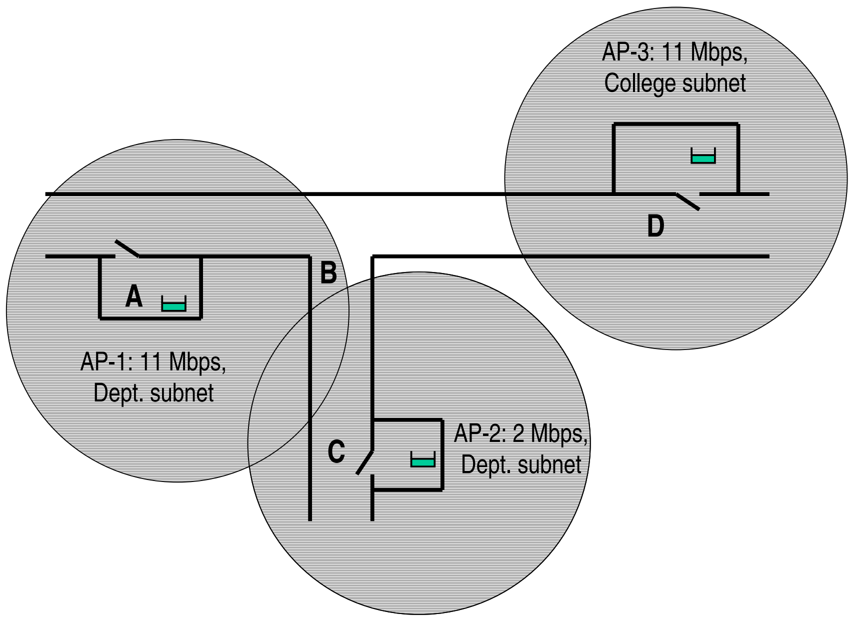


Figure 5.10: The configuration of the access points used in the experiment.

5.5.2 Self-Management and Self-Optimization

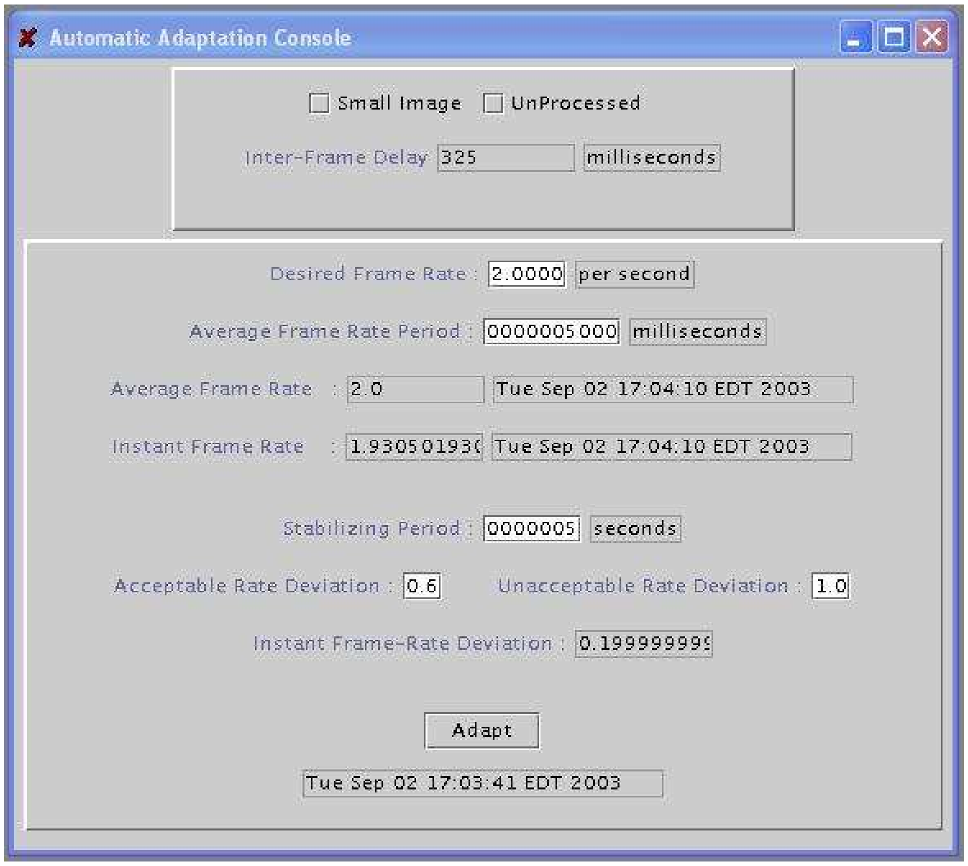
To investigate how ACT/J can support self-management, we developed an application-specific rule that maintains the frame rate of the application by controlling the image size or inserting inter-frame delays dynamically. The original image retrieval application operates in a default mode, which retrieves and plays images as fast as possible. ACT/J enables a developer to weave the rule into the application at run-time, thereby providing new functionality (frame rate control) transparently with respect to the application. The self-optimization rule maintains the frame rate of the application in the presence of dynamic changes to the wireless network loss rate, the network (wired/wireless) traffic, and CPU availability.

Figure 5.12 shows the Automatic Adaptation Console, which diplays the application status and also enables the user to 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`

Figure 5.11: An image from the experiment.

`Period` indicates the period during which the average frame rate should be calculated to be considered for adaptation. The `Stabilizing Period` specifies the amount of time that the rule should wait until the last adaptation stabilizes; also if a sudden change occurs in the environment such as hand-off from one wireless cell to another one, 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.

Figure 5.13 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 Figure 5.12. For the first 60 seconds of the experiment,


Automatic Adaptation Console

Small Image UnProcessed

Inter-Frame Delay 325 milliseconds

Desired Frame Rate : 2.0000 per second

Average Frame Rate Period : 0000005000 milliseconds

Average Frame Rate : 2.0 Tue Sep 02 17:04:10 EDT 2003

Instant Frame Rate : 1.93050193(Tue Sep 02 17:04:10 EDT 2003

Stabilizing Period : 0000005 seconds

Acceptable Rate Deviation : 0.6 Unacceptable Rate Deviation : 1.0

Instant Frame-Rate Deviation : 0.19999999

Adapt

Tue Sep 02 17:03:41 EDT 2003

Figure 5.12: Automatic Adaptation Console.

the user stays close to the location A (Figure 5.10). The rule detects that the desired frame rate is lower than the maximum possible frame rate, based on observed round-trip times. Hence, it inserts an inter-frame delay of approximately 200 milliseconds to maintain the frame rate at about 2 frames per second. At point 120 seconds, the user starts walking from location A to location B for 60 seconds. The automatic adaptation rule maintains the frame rate by decreasing the inter-frame delay during this period. At point 180 seconds, the user begins walking from location B to location C and back again, returning to location B at 360 seconds. During this period, because the AP-2 access point provides 2Mbps, the automatic adaptation rule detects that the current frame rate is lower than that desired. It first removes the inter-frame delay, but the frame rate does not reach to 2 frames per second. Therefore, it reduces the quality of the image by asking for a smaller image size. Now the frame increases beyond that desired, so the automatic adaptation rule inserts an inter-frame delay of 400 milliseconds to maintain the frame rate at 2 frames per second. Although there is some oscillation, the rate stabilizes by time 360 seconds. At this point, the user continues walking from location B to location A, prompting the rule to reverse the actions. First the inter-frame delay is increased to maintain the frame rate, followed by an increase in image size. In this manner, the rule brings the application back to its original behavior. Again, because the current frame rate is higher that expected, an inter-frame delay of about 200 milliseconds is inserted to maintain the frame rate at 2 frames per second.

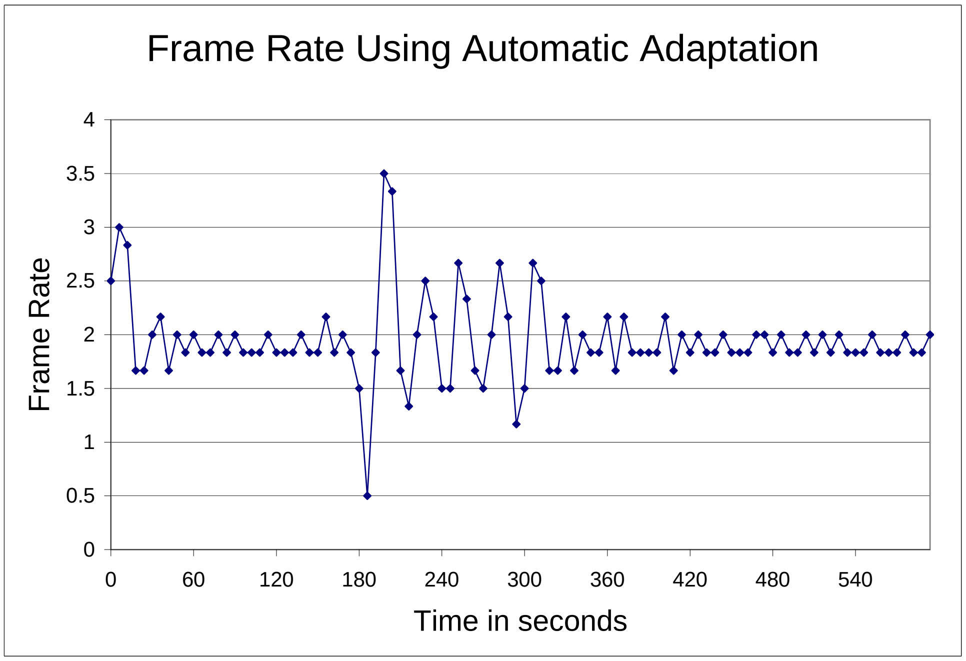


Figure 5.13: Maintaining the application frame rate using automatic adaptation.

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

5.6 Case Study: Coupling ACT and QuO

To investigate the integration of ACT with an existing CORBA framework, we combined ACT/J with the Quality Objects (QuO) framework [42], developed by BBN Technologies and released under an open-source license. QuO is a powerful adaptive framework that supports dynamic adaptability in CORBA and Java RMI applications. 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 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.

5.6.1 QuO Background

QuO employs aspect-oriented programming [1] to separate the non-functional (systematic) aspects from the functional aspects of an application. Figure 5.14 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) [7]. The QuO core comprises a contract and several system conditions. A *contract* is written in the contract-description language (CDL) [7] 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, QuO weaves a qosket into an application at compile time.

As shown in Figure 5.14, 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

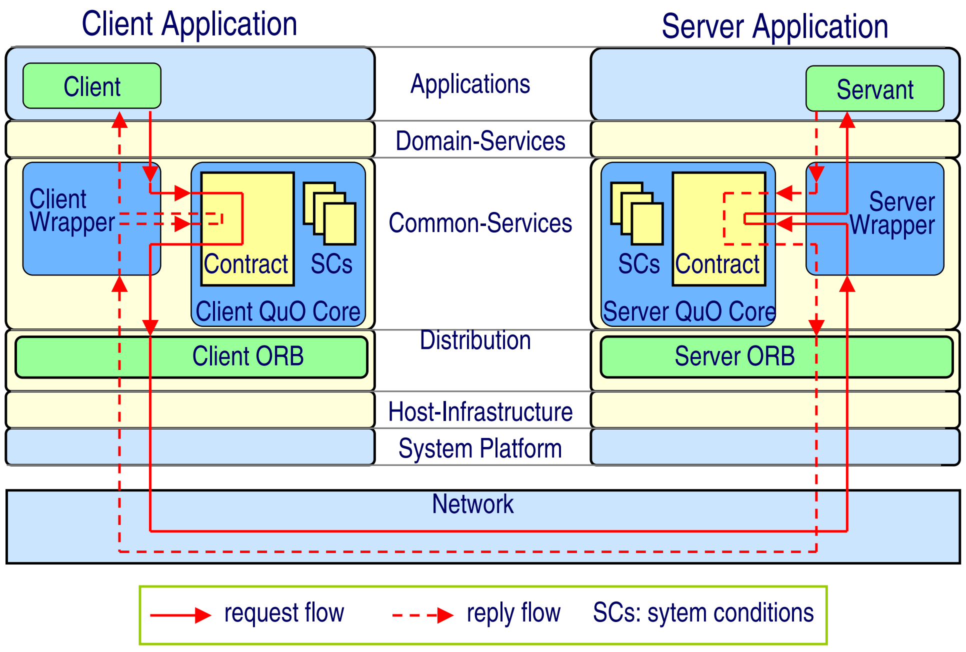


Figure 5.14: A simplified depiction of the QuO architecture.

stub (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.

5.6.2 Dynamic Weaving of Qoskets Using ACT

Combining ACT with QuO enables transparent weaving of new qoskets into applications at run time. 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 can be added to these applications as they execute. Second, embedded applications are required to provide very small footprints; a minimal adaptive core can be compiled with the application, and optional adaptive 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 adapted transparently using ACT and QuO, without modifying or even recompiling the application source code.

Figure 5.15 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.

5.6.3 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 (introduced before) 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

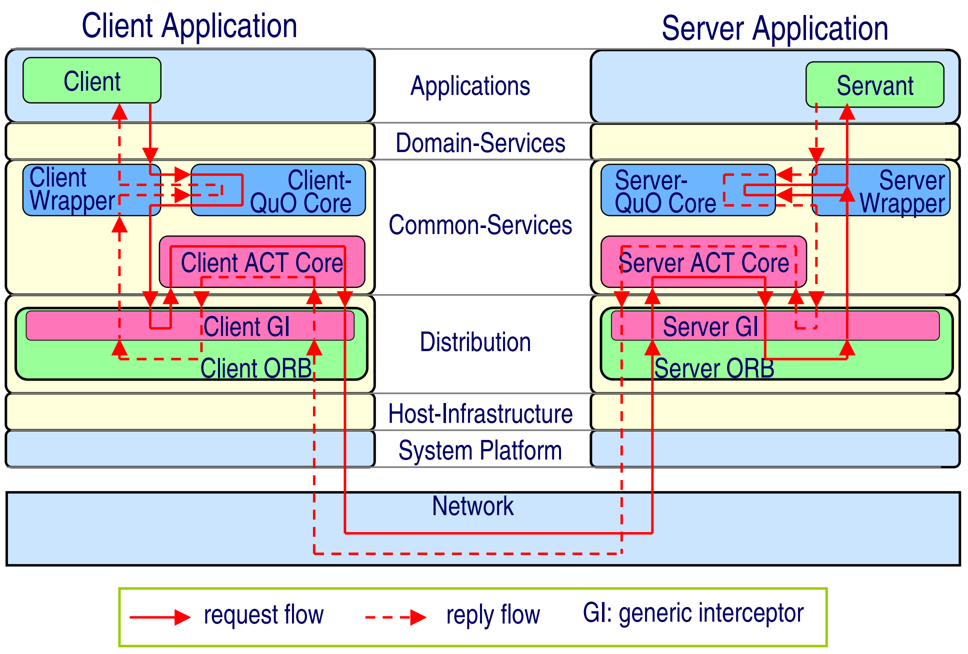


Figure 5.15: Coupling ACT and QuO.

different versions of the images. For example, selecting small instead of large versions of images can be used to reduce bandwidth consumption and delay.

First, we incorporated ACT/J into this application by introducing generic interceptors. To do so, we started the application with a command-line parameter directing it to an ORBacus configuration file defining how to load, create and register a generic interceptor with the application ORB. At this point the application is adapt-ready. Figure 5.16 compares the round-trip delay for retrieving images of varying size, using both the original application and the adapt-ready version. As shown, this overhead is negligible.

Next, 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. Figure 5.17 and 5.18 show the code that define the contract (in CDL) and the wrapper (in ADL) for the new qosket, respectively. 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. As illustrated in Figure 5.18, this control is accomplished by inserting appropriate delays. For the Fast region, we did not

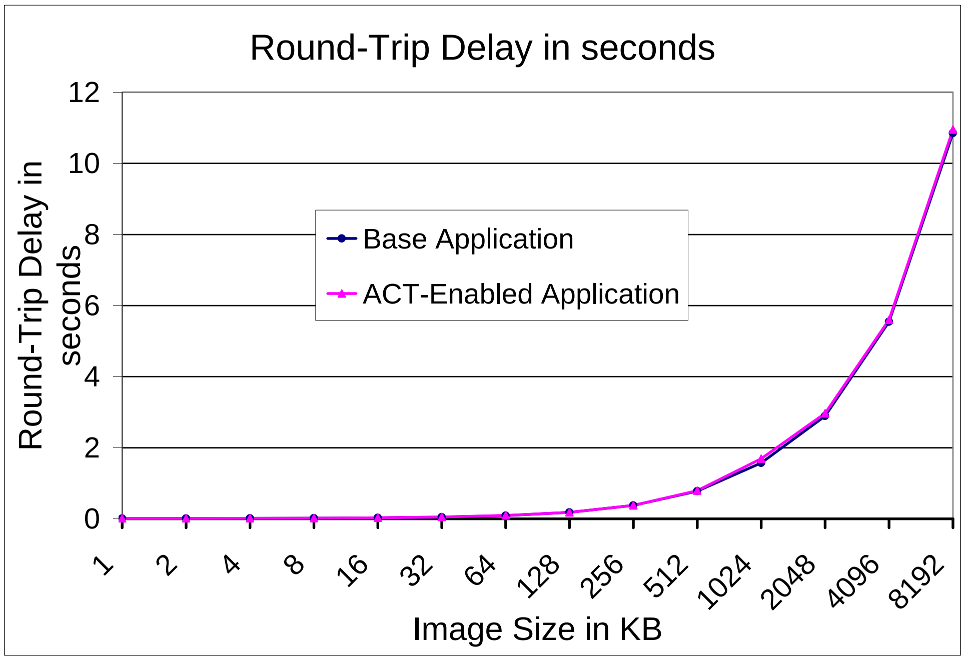


Figure 5.16: Round-trip delay in ACT/QuO application.

insert any delay, but for the Normal and Slow regions, we inserted 50 and 100 milliseconds frame-interval delays, respectively. We used the quogen utility to compile the new qosket.

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

Figure 5.17: Code for the contract of the new qosket written in CDL.

5.6.4 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 (implemented using the image retrieval application) to add a new feature that controls the frame rate. Figure 5.19 shows a sample image from a camera in an instructional laboratory.

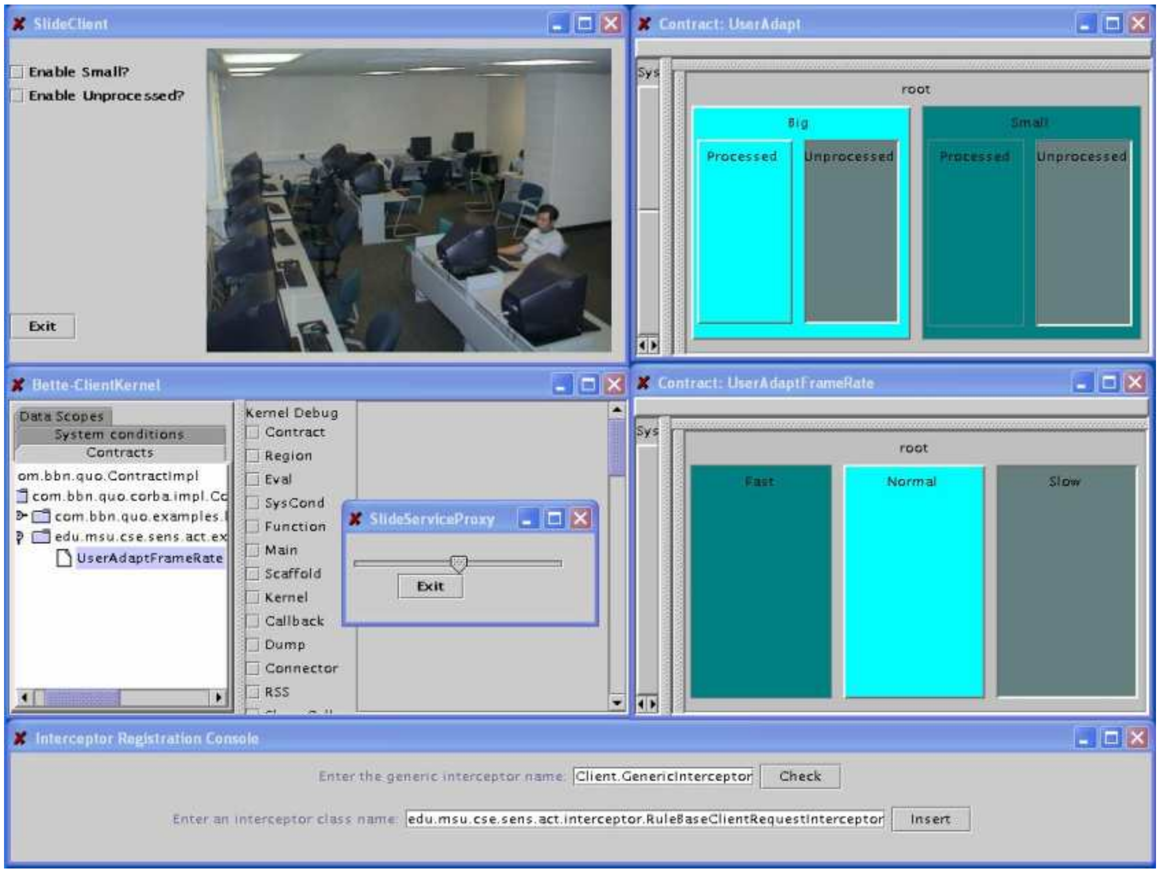
```
behavior UserAdaptFrameRate ()
{
  void slide::SlideShow::read(in long gifNumber,
    out string size, out octetArray buf)
  {
    before METHODCALL
    {
      region Fast {}
      region Normal { ... Thread.sleep(50 ); ... }
      region Slow { ... Thread.sleep(100); ... }
    }
  } ...
}
```

Figure 5.18: Excerpted code for the wrapper of the new qosket written in ASL.

Figure 5.19: Sample image of a monitored instructional laboratory.

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 11Mbps 802.11b wireless network; both systems are running the Linux operating system. At startup 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 "UserAdapt-FrameRate" qosket into the application. Figures 5.20 and 5.21 show screen dumps of the application as it displays large and small versions of an image, respectively.

Figure 5.22 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 application



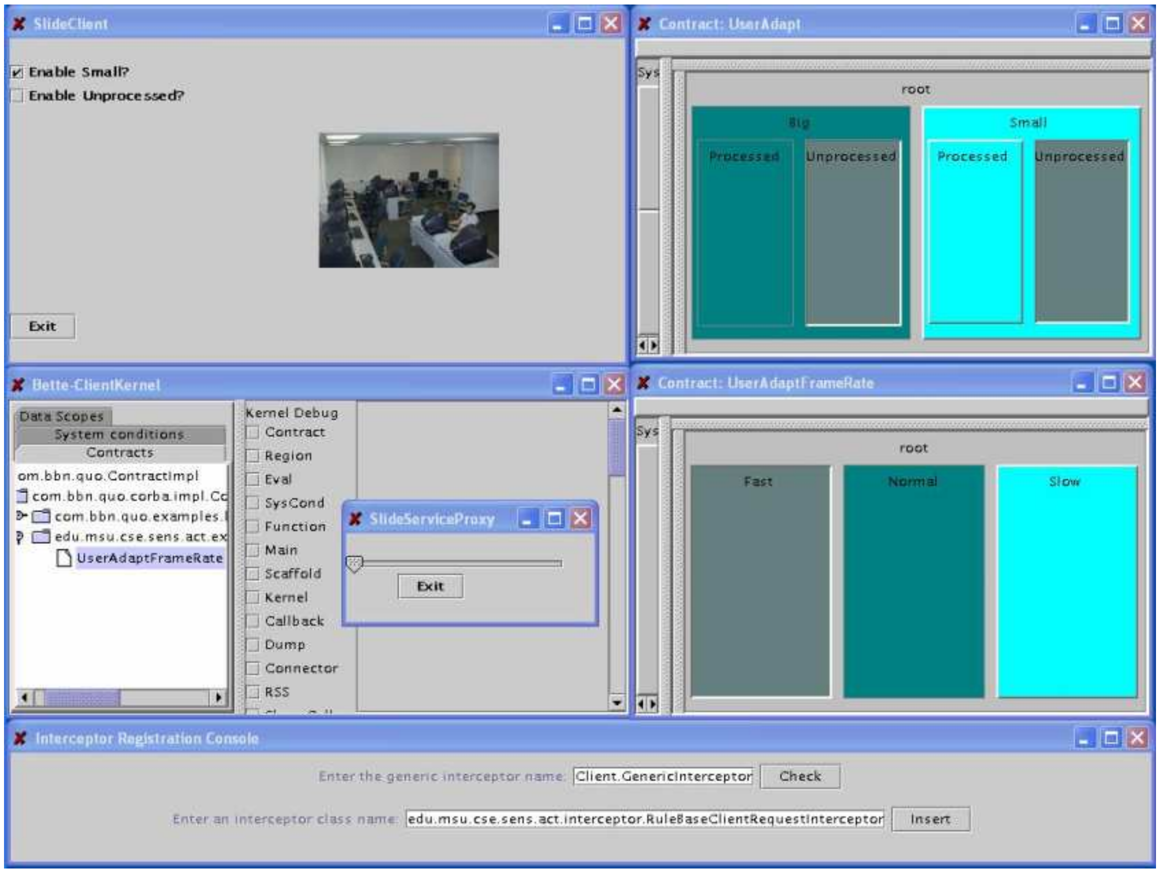
Enter the generic interceptor name: Client.GenericInterceptor

Enter an interceptor class name: edu.msu.cse.sens.act.interceptor.RuleBaseClientRequestInterceptor

Figure 5.20: Screen capture of a 252 KB version of images displayed in the ACT/QuO 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 seconds 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 described in Section 5.4. Figure 5.22 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


☑ Enable Small?
☐ Enable Unprocessed?

Enter the generic interceptor name: Client.GenericInterceptor

Enter an interceptor class name: edu.msu.cse.sens.act.interceptor.RuleBaseClientRequestInterceptor

Figure 5.21: Screen capture of a 19 KB version of images displayed in the ACT/QuO application.

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 seconds into the experiment, the user selects the Normal option; the additional 50 msec delay reduces the frame rate to approximately 7.5 images per second. At 120 seconds, the user chooses the Slow option (100 msec delay), which reduces the frame rate to approximately 5.5 images per second. At 150 seconds, 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 started the application with generic interceptors registered with the

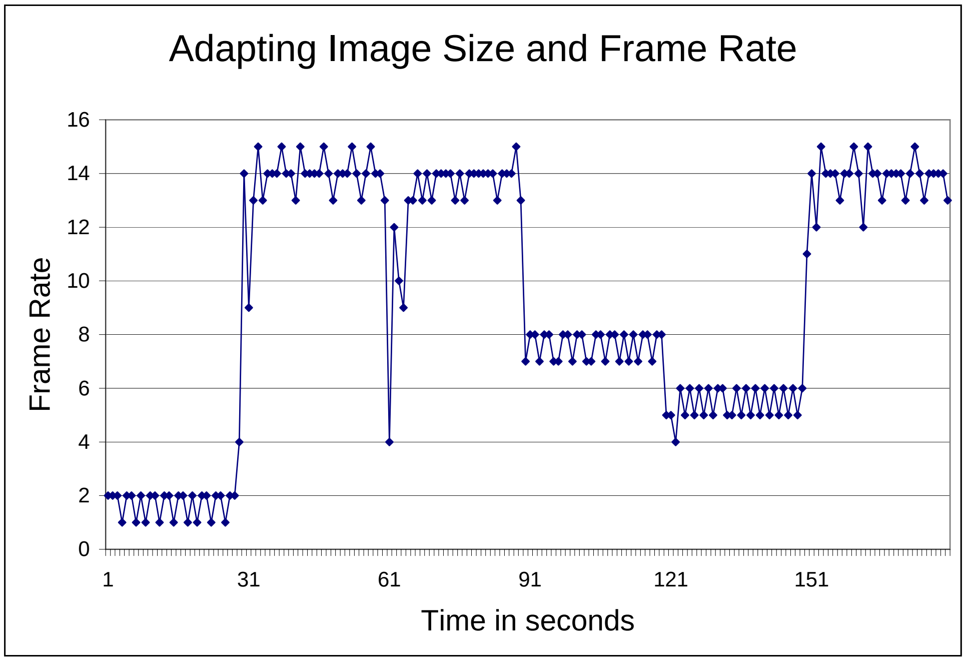


Figure 5.22: Dynamic adaptation in a ACT/QuO hybrid application.

application ORB.

5.7 Related Work

ACT is intended to complement adaptive middleware frameworks and to support interoperation among incompatible 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.

Aspect-Oriented Middleware. Aspect-oriented middleware enables separation of functional aspects from its non-functional aspects (*e.g.*, quality-of-service, security, and fault-tolerance) of a distributed application. One of the most extensive projects in this area is Quality Objects (QuO) [42], 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 intercept requests and replies for possible modifications [42]. In Section 5.6, we showed how ACT can interact with QuO transparently to enable unanticipated adaptation by dynamically weaving new qoskets into the application at run time. In a related project, Jacobsen et al. [166] developed an annotated version of CORBA IDL that enables weaving of semantic properties (such as synchronization and security) into the CORBA skeleton at compile time. AspectIX [28] is an aspect-oriented distribution middleware that is based on the distributed object model [167], 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 [127] 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* [79] transparently so that they could better communicate with Squirrel ORBs.

Reflective Middleware. Reflective middleware uses computational reflection to enables inspection and modification of middleware dynamically during application execution [5]. DynamicTAO [37] and UIC [6] are CORBA-compliant reflective ORBs that employ the component-configurator pattern [75] to support dynamic adaptation. OpenORB [38] is a reflective ORB that provides explicit binding of remote objects and enables unanticipated dynamic adaptation using structural and behavioral reflection [83]. The Coyote project [98] also addresses unanticipated dynamic adaptation in distributed applications using Iguana/J. ZEN [45] is a Java ORB that use Java reflection and the virtual component pattern [112] 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 can host different reflective ORBs transparently while intercepting all CORBA requests, replies, and exceptions and passing them to the appropriate reflective ORB.

Intercepting Middleware. The concept of transparently intercepting CORBA requests and replies has been used in several projects. Friedman et al. [159] use CORBA portable interceptors [47] 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. [116] use portable interceptors to transparently introduce their implementation of fault-tolerant CORBA [47] to CORBA-compliant ORBs. Moser et al. [121] also use an interception-based approach to transparently introduce their implementation of fault-tolerant CORBA (Eternal [121] over Totem [168]) to CORBA applications. Eternal, however, employs an operating-system interception-based approach instead of using CORBA portable interceptors. In ALICE project, Haahr et al. [169] 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, mobility) in ways that were not anticipated at application development time. Moreover, generic interception enables ACT to be used as a framework gateway.

5.8 Summary

In this chapter, we introduced ACT, an extension of transparent shaping 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 unanticipated 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. Two case studies were conducted, where we used ACT/J to accommodate changing conditions of a wireless network infrastructure and to integrate ACT and QuO [42]. 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.

We can use the taxonomy of adaptive middleware introduced in Section 2 to classify ACT. First, ACT is considered in the common services middleware layer because it uses CORBA, which is a distribution middleware, and can be used to implement high-level services such as those defined in CORBA services [170]. For example, ACT can be used to control the QoS in distribution middleware, to apply new security policies at run time, to enable transparent fault tolerance (FT-CORBA can be implemented using ACT), and to perform dynamic type checking. Second, ACT services are accessed by distributed applications transparently; hence, ACT is considered as intercepting middleware. ACT transparently intercepts CORBA requests and modifies them as required. Finally, ACT is considered as tunable middleware because it supports dynamic reconfiguration of distribution services while the core functionality of CORBA is not modified at run time.

Chapter 6

Transparent Application Integration

In the previous chapters, we focused on transparent adaptation for pervasive and autonomic computing. Specifically, in Chapters 4 and 5, we introduced TRAP and ACT as language- and middleware-based approaches to transparent shaping, respectively. In addition, we provided two transparent shaping tools, namely, TRAP/J and ACT/J, to demonstrate the usefulness of transparent shaping in the course of several case studies. In general, the case studies in the previous chapters provided relatively low-level adaptation in single applications. In this chapter, we demonstrate that the same transparent shaping tools can be used beyond a single application and for a higher level type of adaptation, namely, application integration.

To integrate two heterogeneous applications, possibly developed in different programming languages and targeted to run on different platforms, we need to convert data and commands between the two applications on an ongoing basis. The advent of middleware in the 1990's, which hides differences among programming languages, computing platforms, and network protocols [34–36], mitigated the difficulty of application integration. The maturity of middleware technologies resulted in several successful approaches to *enterprise-wide* application integration [171, 172], where applications developed and managed by the same enterprise are made to interoperate with one another.

Ironically, the difficulty of application integration, once alleviated by middleware, has reappeared with the proliferation of *heterogeneous* middleware technologies [173]. As a result, there is a need for a "*middleware* for middleware" to enable Internet-wide and business-to-business application integration [173].

The Web Services Architecture [174] offers one approach to addressing this problem. A *Web service* is a program delivered over the Internet that provides a service described in the Web Service Description Language (WSDL) [175] and communicates with other programs using the SOAP messages [176]. WSDL and SOAP are both independent of specific platforms, programming languages, and middleware technologies. Moreover, SOAP leverages the optional use of HTTP protocol, which allows bypassing firewalls, thereby enabling Internet scalability in application integration.

Although Web services have been successfully used to integrate applications, they do not provide a *transparent* solution to integrate existing applications. The challenge is to integrate existing applications without the need to modify their source code directly. In this chapter, we show how transparent shaping can be used to support transparent application integration.

The rest of this chapter is organized as follows. Section 6.1 provides background on web services. Section 6.2 introduces several alternative architectures supporting application integration. Section 6.3 presents a case study, where we use transparent shaping to integrate two existing applications, one developed in CORBA and the other in the .NET platform. Section 6.4 categorizes research projects and commercial products addressing application integration and discusses how transparent shaping relates to them. Finally, Section 6.5 summarizes this chapter.

6.1 Web Services Background

A service-oriented architecture, as depicted in Figure 6.1, is composed of at least a *provider program*, which is a program capable of performing the actions associated with a service defined in a service description, and a *requester program*, which is a program 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 6.1 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 inter-process communication (IPC).

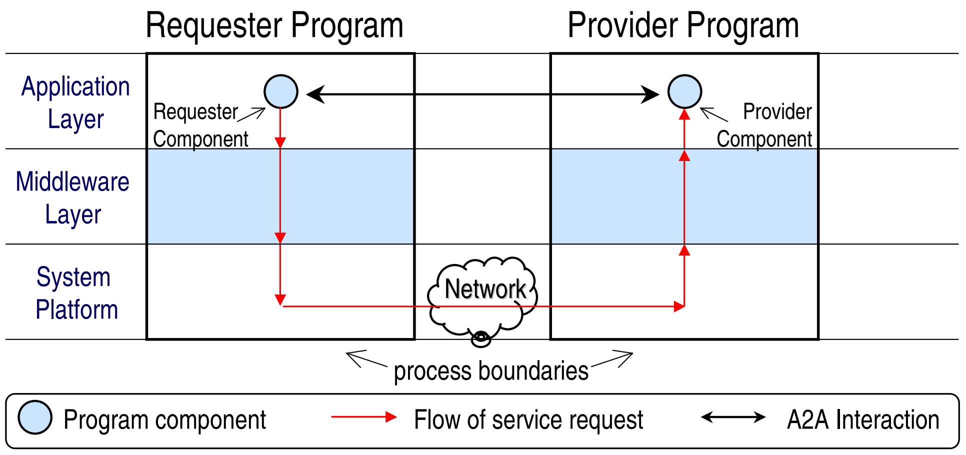


Figure 6.1: A simplified service-oriented architecture.

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 delivered over the Internet that are accessible by other programs using the service descriptor of the Web service defined in WSDL and through the SOAP messaging protocol.

[1]We use the terms “provider program” and “requester program” instead of the terms “provider agent” and “requester agent” used in [174] to avoid the confusion with agents in agent-based systems and to provide consistency with the terms used in the other chapters of this dissertation.

[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 be either implemented using object-oriented languages or designed using an object-oriented paradigm.

SOAP. SOAP [176] is an XML-based messaging protocol independent of specific platforms, programming languages, middleware 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; essentially providing a “middleware for middleware.”

A SOAP message is an XML document with one element, called an envelope, and two children elements, called header and body. The contents of header and body elements are arbitrary XML. Figure 6.2 shows the structure of a SOAP message. The header is an optional element, whereas the body is not optional and 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 [176–178].

```
≺?xml version=“1.0” encoding=“UTF-8” ?≻
≺soap:Envelope xmlns:soap=“http://schemas.xmlsoap.org/soap/envelope/ . . . ≻
  ≺soap:Header≻
    ≺!– Header contents in defined in arbitrary XML. –≻
  ≺/soap:Header≻
  ≺soap:Body≻
    ≺!– Body contents in defined in arbitrary XML. –≻
  ≺/soap:Body≻
≺/soap:Envelope≻
```

Figure 6.2: SOAP message structure.

WSDL. Web Services Description Language (WSDL) [175, 179] 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 service requesters and providers to perform a successful interaction. WSDL addresses this issue by describing an interface to a Web service and providing the convenience of remote procedure calls (or even more complicated interactions such as back-and-forth interactions). For more information about details of WSDL, please refer to [175, 178, 179].

6.2 Transparent Shaping and Application Integration

Several different approaches have been employed in the literature to integrate applications [180]. Regardless of what approaches are being employed, to integrate two heterogeneous applications, essentially we need to translate the syntax and semantics of the two applications during execution. In the rest of this section, we first introduce alternative architectures for application integration and describe how transparent shaping can be used to provide *transparent* application integration. Next, we discuss the role of Web services in the process of application integration.

6.2.1 Alternative Architectures for Application Integration

Depending on *where* the translation is performed (*e.g.*, inside the requester program, inside the provider program, or inside a separate program), we distinguish three approaches to application integration as follows.

Hosting translator components inside a bridge program. An intuitive approach to integrate two applications is to use a *bridge program*, which sits between the two programs, intercepts all the interactions, and translates the interactions from one application semantic and syntax to the other. The architecture for this approach is illustrated in Figure 6.3. The bridge program hosts a *translator component*, which encapsulates 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. We note that a translation may involve more than one requester and provider components and it may not be as simple as a one-to-one mapping of requester and provider components.

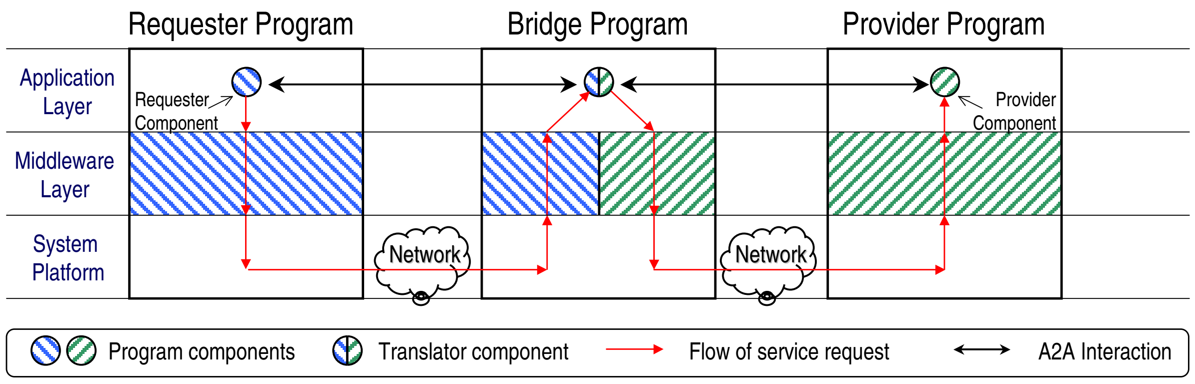


Figure 6.3: Hosting translator components inside a bridge program.

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 in the bridge program once, which will be effective to all the translator components hosted by the bridge.

The main disadvantage of this architecture is the overhead imposed to the interactions because of one extra level of process-to-process redirection (in case the bridge program is located on the same machine as the requester and/or provider programs) or machine-to-machine redirection (in case the bridge program is located on a separate machine). The single-point-of-failure and the bottleneck problem are other disadvantages of this approach.

Hosting Translator Components inside the Requester Program. To avoid the overhead of the extra level of process-to-process or machine-to-machine redirection imposed by the previous architecture, the translator component could instead be hosted inside the requester program, as illustrated in Figure 6.4. However, *transparent* interception and redirection of interactions to the translator component in this approach are not as simple as in the previous approach.

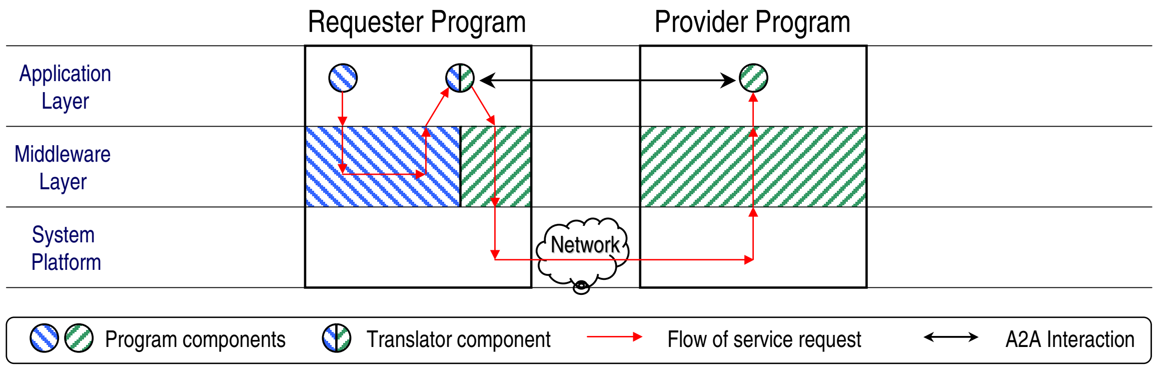


Figure 6.4: Hosting the translator components inside the requester program.

Transparent shaping can be used to provide a transparent application integration by transparently augmenting existing applications with hooks intercepting and redirecting the interactions to adaptive code, which implements the translator. The hooks can be inserted into the application code using a language-based approach such as TRAP or inserted into the supporting middleware using a middleware-based approach such as ACT.

Hosting Translator Components inside the Provider Program. Figure 6.5 shows the architecture of heterogeneous application integration where the provider program hosts the translator. Hosting a translator component inside a provider program is beneficial if several requester programs use the same translator. Instead of modifying all requester programs to host the translator, only the provider program can be modified for this purpose. However, if the translation process is CPU-intensive, this approach may not scale well with the number of requester programs.

As in the previous approach, transparent shaping can be used to host the translator components inside the provider program in a non-invasive manner. However, depending on the specific middleware technologies and the programming languages used to develop requester and provider programs, also depending on the tools available in transparent shaping, it might be easier to host translator components inside the requester program than

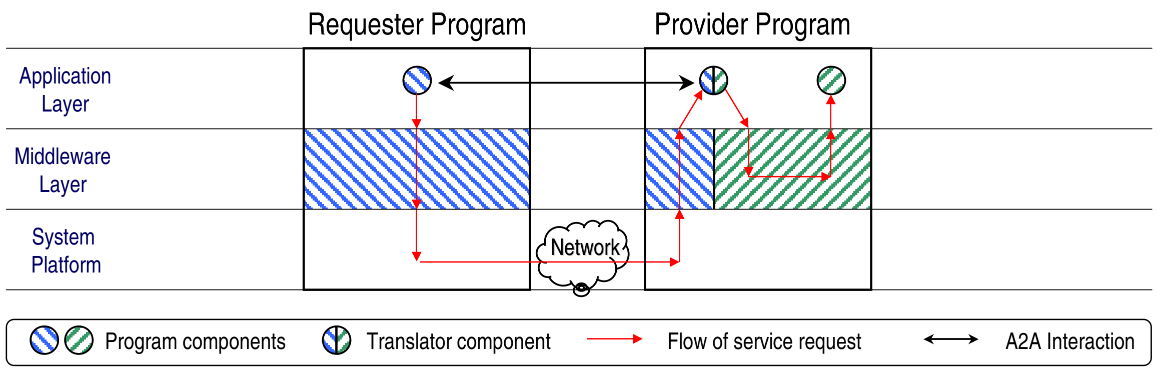


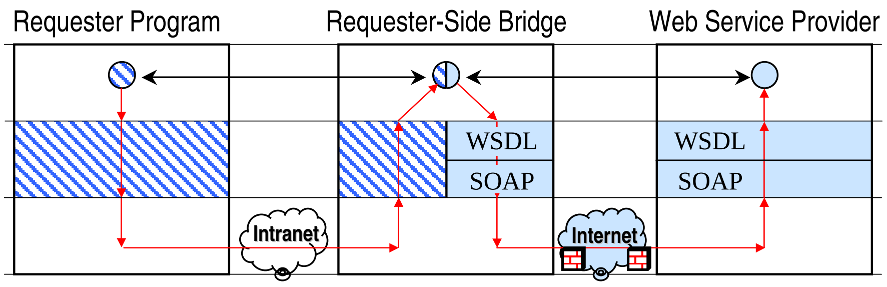
Figure 6.5: Hosting the translator components inside the provider program.

inside the provider program. For example, let us assume that we want to integrate a requester program developed in Java and a provider program developed in C++. Since the tools currently provided in transparent shaping, namely, TRAP/J and ACT/J, support only Java programs, we can only shape the requester program transparently.

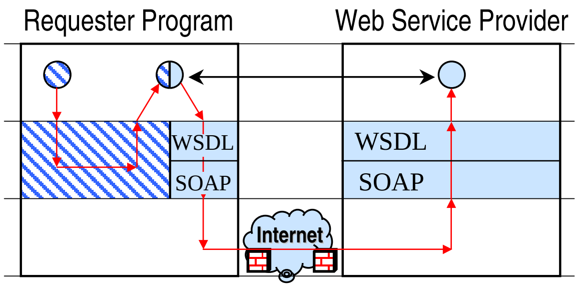
6.2.2 The Role of Web Services in Application Integration

Providing direct translations for N heterogeneous middleware technologies requires N^2 translators to cover all possible application integrations. Using a common language reduces the number of translators from N^2 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.

Figure 6.6(a) and 6.6(b) show two architectures enabling a requester program to use a Web service by hosting the translator component either inside a requester-side bridge or inside the requester program, respectively. Similarly, Figure 6.7(a) and 6.7(b) show two architectures enabling a provider program to be exposed as a Web service by hosting the translator component either inside a provider-side bridge or inside the provider program, respectively. To integrate requester and provider programs, none of which is a Web service requester or provider, we can use a combination of architectures in Figures 6.6(a) and 6.6(b) and Figures 6.7(a) and 6.7(b). Out of the four possible combinations, Figure 6.8(a) and 6.8(b) illustrate only two of them. These two architectures enable integration of two heterogeneous applications through Web services by hosting the translator components either inside requester- and provider-side bridge programs or inside the requester and provider programs, respectively.



(a) Using a requester-side bridge.



(b) Shaping the requester.

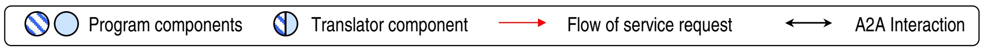
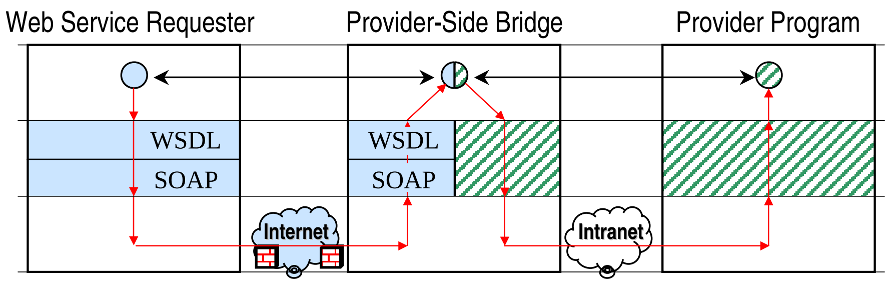


Figure 6.6: Alternative requester-side architectures for application integration through Web services.

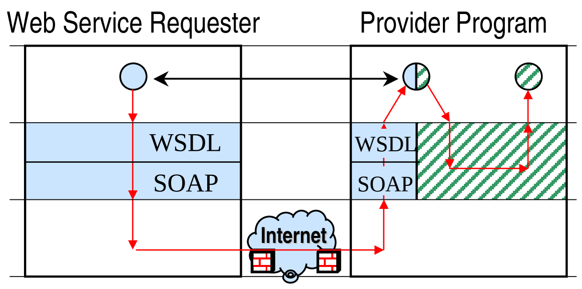
Many existing distributed applications have been developed in heterogeneous platforms such as Java RMI [68], CORBA [47], and .NET [70]. Appendix A provides a complete solution for transparent integration of such heterogeneous applications using transparent shaping and Web services in the course of a simple stock quote example.

6.3 Case Study: Integrating Two Existing Applications

To show how transparent shaping can be used to integrate *existing* applications transparently, in this section we integrate an existing CORBA application with an existing .NET



(a) Using a provider-side bridge.



(b) Shaping the provider.

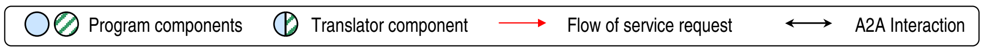
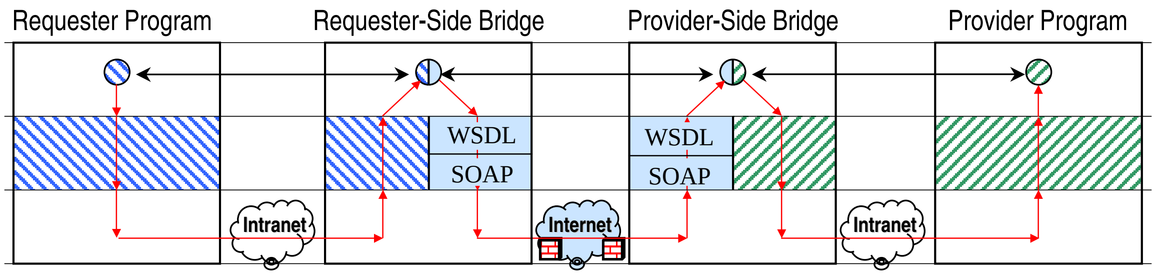


Figure 6.7: Alternative provider-side architectures for application integration through Web services.

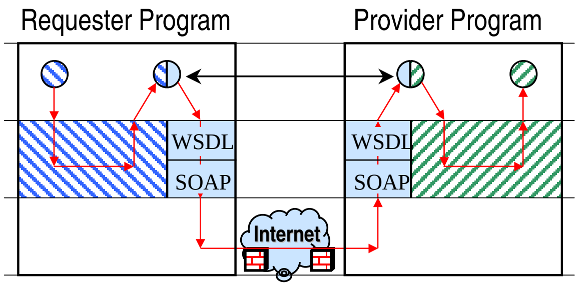
application through Web services. The architecture that we use for this integration is the one illustrated in Figure 6.8(b), where the Web service translators are hosted inside the requester and provider programs. In the remainder of this section, we first introduce each of the two applications briefly. Next, we describe our strategy for how to shape each application to interoperate with Web services. Finally, we describe the details of the shaping process.

6.3.1 The Image Retrieval Application

The first application is a distributed image retrieval application developed by BBN Technologies distributed with the QuO framework [42]. We previously introduced and used this application in Chapter 5. The application is a CORBA application developed in Java. It



(a) Using requester- and provider-side bridges.



(b) Shaping both the requester and provider.

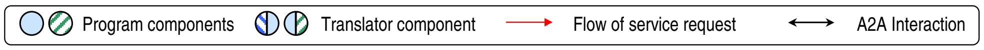


Figure 6.8: Two combinations out of four possible combinations resulting from combining Figures 6.6(a) and 6.6(b) and Figures 6.7(a) and 6.7(b).

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 image retrieval application by itself can benefit from the QuO framework, which supports several adaptive behaviors. However, in this study we disabled the QuO framework and used the application only as a CORBA application. A screen dump of the client program GUI is depicted in Figure 6.9, which shows an aerial photograph retrieved from the server program. The client program continuously sends requests to the server program asking for images. After each request is replied, the retrieved image is displayed.

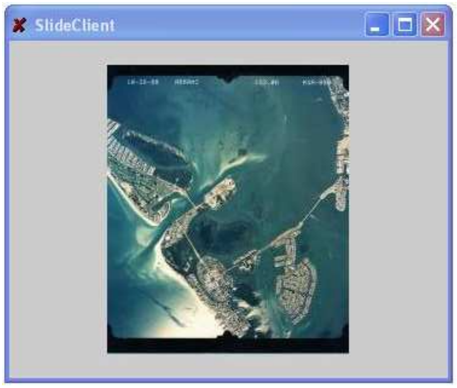


Figure 6.9: A screen dump of the client program GUI showing an aerial photograph of the Sarasota bay.

6.3.2 The Sample Grabber Application

The second application is a sample grabber application (called `SampleGrabberNET`) that is part of the `DirectShow.NET` framework developed by NETMaster[3]. This application is a .NET application written in C# and is freely available at the Code Project web site (URL: `http://www.codeproject.com/`). It uses the interfaces provided in the `DirectShow.NET` framework to interoperate with DirectShow.

DirectShow [181] is a standard Microsoft Win32 API that can be used from a Windows application to interact with compliant movie and video devices installed on a Windows computer. DirectShow is developed as COM components [71] and can be used through COM programming in a Visual C++ program. The `DirectShow.NET` framework by NETMaster enables a convenient use of DirectShow in C#. All the DirectShow interfaces written in IDL is rewritten in C# that are compliant with the DirectShow documents for Visual C++ provided by Microsoft.

Figure 6.10 shows a screen dump of the frame grabber application GUI. On the left side of the GUI, a preview panel shows a live video stream captured from a video camera

[3]NETMaster is an active member of the Code Project. The Code Project (URL: `http://www.-codeproject.com/`) is a place for a large number of free C++, C# and .NET articles, code snippets, discussions, and news on the Internet. It organizes the papers and programs developed by its members and provides them freely to be used or improved by others.

installed on a Windows XP machine. On the right side, an image panel shows a bitmap image grabbed from the camera using the “Grab” toolbar button.

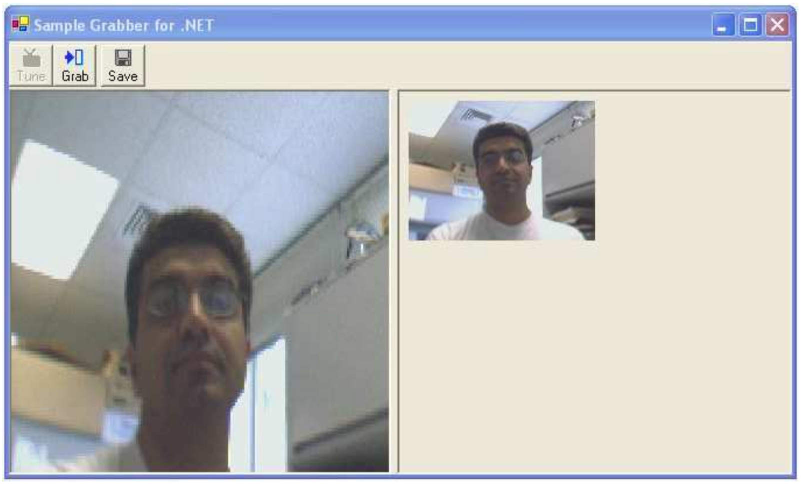


Figure 6.10: A screen dump of the sample grabber application GUI. The left panel shows a preview of the live video captured from a video camera and the right panel shows a still image grabbed using the “Grab” toolbar button.

Similar to other DirectShow applications, the frame grabber application first builds a *filter graph* and then controls the filter graph and responds to the events fired in the graph. The filter graph for the sample grabber application is illustrated in Figure 6.11. Basically, a filter graph is a directed graph of filters, which are the basic building blocks of DirectShow applications and generally perform a single operation on a multimedia stream. For example, a filter may read multimedia files, capture video from a video capture device, encode or decode a particular stream format (*e.g.*, MPEG-1 video), or send data to a graphics or a sound card.

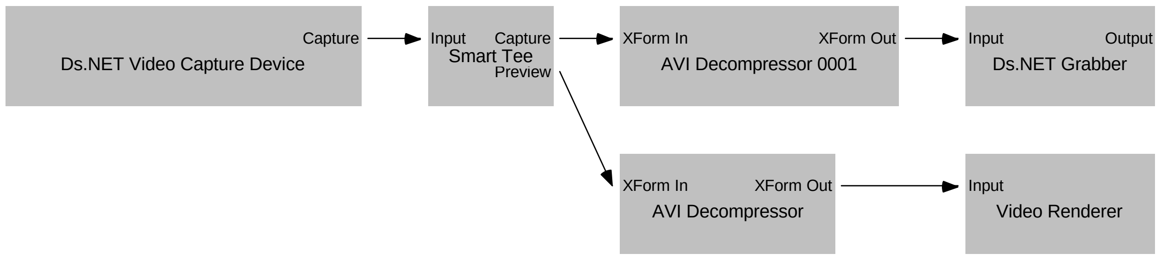


Figure 6.11: The filter graph of the .NET frame grabber application taken by the GraphEdit tool.

The filter graph first captures a live video stream from a video capture device using the `Ds.NET Video Capture Device` filter. Next, it makes two copies of the video stream using the `Smart Tee` filter and decompresses them using two `AVI Decompressor` filters. Finally, it provides a preview of the video stream using the `Video Renderer` filter and enables frame grabbing using the `Ds.NET Grabber` filter. Once the "Grab" button is pressed, the request is handled by grabbing a frame using the `Ds.NET Grabber` filter.

GraphEdit [181] is one of the standard tools distributed with Microsoft DirectShow. Using GraphEdit, we can discover the graph used in a DirectShow application while it is running without the need to look into the application source code. Figure 6.11 shows the the filter graph of the sample grabber application that is obtained by the GraphEdit tool.

6.3.3 Application Integration Strategy

Our goal is to enable the client program in the CORBA image retrieval application to retrieve live images from the .NET frame grabber application. To make this possible, we need to shape both applications to interoperate with each other. As described in Section 6.2, there are several architectures that we can choose for this application integration. Also, as described in Sections A.2, A.3, and A.4, there are several solutions to implement each architecture.

Among the architectures, we selected the one illustrated in Figure 6.8(b), which is the result of combining the architectures illustrated in Figures 6.6(b) and 6.7(b). This architecture provides an application integration through the use of Web services, where the Web services translator components are hosted inside the requester and provider programs. 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 application plays the role of a requester program and must be shaped to use the frame grabber Web service.

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

6.3.4 Exposing the Frame Grabber Application as a Web Service

As explained in Section A.4.2, we have two solutions to expose a .NET server program as a Web service using the architecture in Figure 6.7(b). The first solution uses an IIS Web server to host the translator and provider components, while the second solution uses the .NET server program itself as a Web service. The latter solution is better than the former, if the none of the types to be exposed by the .NET server program is a .NET specific types [182, 183]. Since we do not need to use any of the .NET specific types to expose the frame grabber application as a Web service, we pick the second solution.

However, the 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. So, we first need to shape the .NET frame grabber application to become a .NET remoting application and then use it as a Web service. As we do not have a C# version of the TRAP generator framework, we cannot do this part transparently from the application source code. We note that when the TRAP/C# becomes available, this part can be done automatically (*i.e.*, there will be no need to directly modify the source code of the .NET frame grabber application).

Our goal is to minimize the modifications to the frame grabber application source code. Therefore, we only put a hook inside the application and put the rest of the code regarding making the application as a .NET remoting application in a separate program. These two programs are loaded inside another program, called `Shape.exe`, which is listed in Figure 6.12 (lines 1 to 14). The modified .NET frame grabber program is inside the `SampleGrabberNET.exe` assembly[4], the .NET remoting code is inside the `DotNETServer.exe` assembly, and the configuration file for the `Shape.exe` is inside the `Shape.exe.config`

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

file. The excerpted code for the `Shape.exe.config` configuration file is listed in Figure 6.12 (lines 16 to 23).

```
 1  // The host application defined in Shape.cs
 2  public class Shape {
 3    static private string configFilename, dotNETServer, sampleGrabberNET;
 4    public static void Main(string [] args) {
 5    if (args.Length != 3) return;
 6    configFilename = args[0]; dotNETServer = args[1]; sampleGrabberNET = args[2];
 7    try {
 8      RemotingConfiguration.Configure(configFilename);
 9      AppDomain ad = AppDomain.CurrentDomain;
10      ad.ExecuteAssembly(dotNETServer);
11      ad.ExecuteAssembly(sampleGrabberNET);
12    } catch(Exception e) {}
13    String keyState = “”; keyState = Console.ReadLine();
14  }
15
16  // The configuration file defined in Shape.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 6.12: Excerpted code for the Shape program that hosts both the .NET server and frame grabber assemblies.

The command line that we use to run the provider program is the following: `Shape.exe Shape.exe.config DotNETServer.exe SampleGrabberNET.exe`. As listed in Figure 6.12, first, the configuration file is parsed and the instructions are followed (line 8), which provides flexibility to configure the `Shape` program at startup time as discussed before. Next, the `DotNETServer.exe` and the `SampleGrabberNET.exe` are executed using the .NET reflection facilities (lines 10 and 11).

To put a hook inside the `SampleGrabberNET.exe` assembly, we added one *public* method to the frame grabber application that basically grabs a new frame and returns it as a bitmap image. Figure 6.13 lists all the code that we directly added to the sample grabber application (the added code is in *italic*). The hook is the `public Image grab-`

`Sample_WovenCode()` method inside the `MainForm` class (lines 8 to 16). Basically, this method calls the `toolBar_ButtonClick()` method (line 11) to grab a frame. By using the codetoolBar_ButtonClick() method to grab a frame, we make the calls to the hook method appear to the original application as if the user has clicked on the "Grab" button.

```
 1  // The MainFormcs file used in the SampleGrabberNET.exe program
 2  using System.Threading;
 3  namespace SampleGrabberNET {
 4  public class MainForm : System.Windows.Forms.Form, ISampleGrabberCB {
 5    private object lockObject = new Object();
 6    private Image image = null;
 7    private Bitmap bmp = null;
 8    public Image grabSample_WovenCode() {
 9      lock(lockObject) {
10        image = null;
11        toolBar_ButtonClick(this, new ToolBarButtonClickEventArgs(toolBarBtnGrab));
12        if ( image == null) try { Monitor.Wait(lockObject); } catch ( . . . ) { . . . }
13        Monitor.Pulse(lockObject);
14      }
15      return bmp;
16    }
17
18    void OnCaptureDone() { . . .
19      lock(lockObject) {
20        Image o = bmp; bmp = new Bitmap(b);
21        if( o != null ) o.Dispose();
22        Monitor.Pulse(lockObject);
23      }
24    }
25
26    static private MainForm instance = null;
27    static public MainForm getInstance() {
28      if (instance == null) instance = new MainForm();
29      return instance;
30    }
31    [STAThread] static void Main() {
32      // Application.Run(new MainForm());
33      Application.Run(getInstance());
34  } }
```

Figure 6.13: Direct modifications to the .NET frame grabber application source code.

Unfortunately, implementing the hook was not as easy as we expected. Grabbing a frame in a DirectShow application is done through the use of a filter graph, which works asynchronously with respect to the thread executing the DirectShow application [181]. In other words, the thread that asks for a frame does not block until the frame is grabbed and returned by the filter graph. Instead, the filter graph will send an event to the application when the frame is ready to be returned. The event is handled in the original program inside the `OnCaptureDone` method. To ensure that the thread calling the hook method is notified when the frame is ready, we used a lock object kept in the `lockObject` variable (line 5). The calling thread waits on the lock object (line 12) until the `OnCaptureDone()` method is called as a result of frame being ready. The lock object is pulsed inside the `OnCaptureDone()` method to notify the calling thread that the frame is ready to be returned (line 22).

Finally, we need to enable the code inside the `DotNETServer.exe` assembly to be able to get a hold of the hook inside the `SampleGrabberNET.exe` assembly. For this purpose, we made the `MainFrame` class as a singleton class [59] (lines 26 to 30 and 32 to 33). In this way, the instance of the `MainForm` class inside the `SampleGrabberNET.exe` will be accessible to the code inside the `DotNETServer.exe`.

Figure 6.14 shows the excerpted code of the `SampleGrabberObject` class defined in the `SampleGrabberObject.cs` file and used in the `DotNETServer.exe` program. First, the .NET reflection facilities is used to get a hold of the hook, which is the `public Image grabSample_WovenCode()` method (lines 3 to 10). Specifically, the (getInstance()) method of the `MainForm` singleton class is used to get a reference to the singleton object of this class, which is kept in the `mf` variable (line 8). The `mf` variable is used to get a reference to the hook, which is kept in the `mi` variable (line 9).

Next, the `public short[] GrabFrame( int nQuality )` method is defined (lines 11 to 22), which is the method that is exposed by the `DotNETServer.exe` program that can be used from a .NET client application or a Web service requester program (described next). This method first calls the hook method, which gets a live frame from the camera and returns a `Bitmap` image (lines 12 to 13). Next, it converts the `Bitmap` image to a `jpeg` image using the `nQuality` (lines 14 to 20). `nQuality` can vary from 0

```
 1  // The SampleGrabberObject.cs file used in the DotNETServer.exe program
 2  public class SampleGrabberObject : MarshalByRefObject {
 3    private MethodInfo mi; private Object mf = null;
 4    public SampleGrabberObject() {
 5      AppDomain ad = AppDomain.CurrentDomain;
 6      Assembly [] assemblies = ad.GetAssemblies();
 7      // Finding the main form using the .NET reflection facilities (not shown).
 8      mf = mainForm.InvokeMember(”getInstance”, . . . );
 9      mi = mf.GetType().GetMethod(”grabSample_WovenCode”);
10    }
11    public short[] GrabFrame( int nQuality ) {
12      object [] parameters = new object[0];
13      Bitmap bitmap = (bitmap)mi.Invoke( mf, parameters );
14      ImageCodecInfo myImageCodecInfo = GetEncoderInfo(”image/jpeg”);
15      EncoderParameters encps = new EncoderParameters( 1 );
16      EncoderParameter encp = new EncoderParameter( Encoder.Quality, (long)nQuality);
17      encps.Param[0] = encp;
18      MemoryStream ms = new MemoryStream();
19      bitmap.Save( ms, myImageCodecInfo, encps );
20      ms.Close();
21      return converyByteArray2ShortArray( ms.ToArray() );
22    }
23  }
```

Figure 6.14: Shape.

to 100, where `nQuality=100` means to compress the image with 100% quality. Finally, it converts the `byte[]` data to the `short[]` for compatibility reasons (`unsignedByte` defined in the XML schema data types is not interpreted in the same way in the C# and Java languages) and returns the image (line 21).

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 `SOAPsuds.exe` utility with the `-sdl` option that generates a WSDL schema file. The excerpted WSDL description is listed in Figure 6.15. This WSDL describes an *abstract* application-level service description (interface) to the Web service (lines 3 to 16) as well as a *concrete* protocol-dependent details of how to access the service (lines 18 to 33).

The abstract description part (lines 3 to 16) describes the interface to the Web service using the `message` elements (lines 3 to 8), which defines what type of messages can be sent

```
 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 6.15: The excerpted WSDL description of the sample grabber Web service.

to and received from the Web service, and the `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 description part (lines 18 to 33) complements the abstract part using the `binding` element (lines 18 to 26), which basically describes *how* a given interaction is performed over *what* specific transport protocol, and the `service` element (lines 28 to 33) that describes *where* to access the service. The how part describes how marshaling and unmarshaling 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).

6.3.5 Transparent Shaping of the Image Retrieval Client Program

According to our strategy, we follow the architecture illustrated in Figure 6.6(b) to shape the CORBA client program to interoperate with the .NET frame grabber program that is exposed as a Web service provider program. As described in Section A.2, we have two solutions that can be used to host the translator component inside the CORBA client programs. Among the two solutions, we picked the first solution, where we use the ACT framework (introduced in Chapter 5) to host a proxy object that plays the role of the Web service translator for the client program.

To intercept and redirect the CORBA requests, first, we make the client program adapt-ready by running the program using two extra command-line parameters: `java client` *`ORBconfig file:client.cfg`*.[5] Next, we insert a new rule to the rule-based decision maker of the ACT core that intercepts all the CORBA requests.

Figure 6.16 lists the excerpted code of the condition and action classes of the rule. The condition part of the rule is defined in the `SlideService_Condtion.java` file (lines 1 to 8) that returns `true` always to make all the intercepted CORBA request to be forwarded to the action part of the rule. The action part of the rule is defined in the `SlideService_Action.java` file (lines 10 to 25). In the constructor of the `Slide-Service_Action` class (lines 12 to 17), an instance of the translator component (defined in the `SlideService_ClientLocalProxy.java` file, which is described next) is created.

Once the rule is inserted, all CORBA requests will be reified by the CORBA ORB and will eventually be intercepted by the `process()` method of the `SlideService_Action`

[5]For details of how ACT works, please refer to Chapter 5.

```
 1   // The condition class defined in SlideService_Condtion.java
 2   public class SlideService_Condition extends InBandDMCondition {
 3     public SlideService_Condition(ActiveRule activeRule, ORB orb) { super(...); }
 4     public boolean check(org.omg.CORBA.Object targetObj, FullInterfaceDescription
 5       fullIntDesc, ServerRequest serverRequest, Request request) {
 6       return true;
 7     }
 8   }
 9
10   // The action class defined in SlideService_Action.java
11   public class SlideService_Action extends edu.msu.cse.sens.act.dm.InBandDMAction {
12     public SlideService_Action(ActiveRule activeRule, org.omg.CORBA.ORB orb) {
13       super(activeRule, orb);
14       SlideService_ClientLocalProxy slideService_ClientLocalProxy =
15         new SlideService_ClientLocalProxy(orb);
16       // publishing the SlideService_ClientLocalProxy CORBA object in the naming service
17     }
18   public boolean process(org.omg.CORBA.Object targetObj, FullInterfaceDescription
19     fullIntDesc, ServerRequest serverRequest, Request request) {
20     request = createReq(slideService_ClientLocalProxy, serverRequest, fullIntDesc, opNum);
21     request.invoke();
22     org.omg.CORBA.Any res_any = request.result().value();
23     serverRequest.set_result(res_any);
24     return true;
25   }
```

Figure 6.16: Excerpted code for the condition and action classes for shaping the CORBA image retrieval client program using the ACT framework.

class (lines 18 to 25). The `process()` method creates another CORBA request similar to the one intercepted, except that its target object is `slideService_ClientLocalProxy`.

The `SlideService_ClientLocalProxy` class is defined in the `SlideService_ClientLocalProxy.java` file that is listed in Figure 6.17. 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 listed in Figure 6.15. Next, all calls to the original CORBA object are forwarded to the Web service (lines 14 to 27).

Figure 6.18 lists the IDL description used in the original of the CORBA image retrieval application. The `SlideShow` interface defines six methods (lines 4 to 9). As listed in Figure 6.17 (lines 21 to 26), all the `read*()` methods defined in the IDL file are mapped

```
 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 readBig(int gifNum, StringHolder sizeHolder, octetArrayHolder pixHolder) {
22      pixHolder.value = grabFrame( 75 ); sizeHolder.value = "big";
23    } . . .
24    public void readSmall(int gifNum, StringHolder sizeHolder, octetArrayHolder pixHolder) {
25      pixHolder.value = grabFrame( 25 ); sizeHolder.value = "small";
26    } . . .
27    public int getNumberOfGifs() { return -1; }
28  }
```

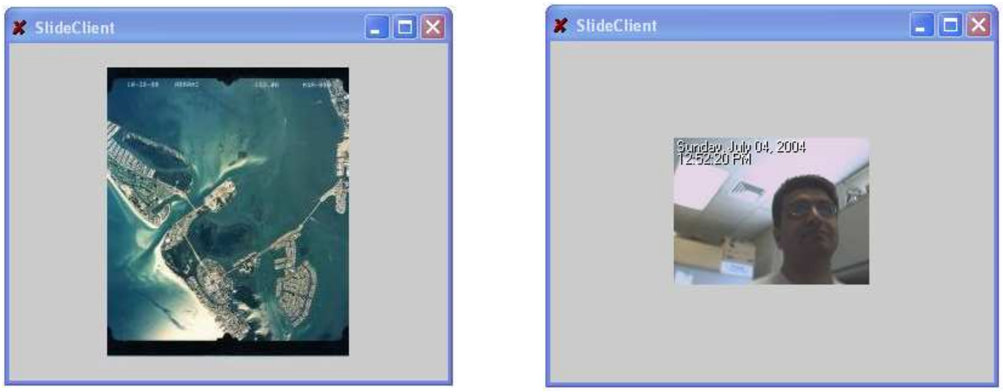
Figure 6.17: Excerpted code for the Web service translator component defined as a proxy object in the ACT framework.

to the `GrabFrame()` method of the Web service exposed by the provider program. The `getNumberOfGifs()` method simply returns -1 (line 27) to indicate that the images being retrieved are live images (as opposed to being retrieved from a number of stored images at the server side).

Figure 6.19 depicts two screen dumps of the GUI of the CORBA image retrieval program. Figure 6.19(a) depicts the client application while it is using the CORBA server application, where the image shows a stored aerial image. Figure 6.19(b) depicts the client application after it has been shaped dynamically to use the frame grabber Web service, where the captured image shows a live picture of a user from the camera installed at the machine running the provider program.

```
// The slide show interface defined in SlideShow.idl
module com { module bbn { module quo { module examples { module bette {
  interface SlideShow {
    void readSmall ( in long gifNumber, out string size, out octetArray buf );
    void readSmallProcessed ( in long gifNumber, out string size, out octetArray buf );
    void readBig ( in long gifNumber, out string size, out octetArray buf );
    void readBigProcessed ( in long gifNumber, out string size, out octetArray buf );
    void read ( in long gifNumber, out string size, out octetArray buf );
    long getNumberOfGifs ( );
  };
}; }; }; }; };
```

Figure 6.18: The slide show IDL file.



(a) Before integration.

(b) After integration.

Figure 6.19: Two screen dumps of the CORBA image retrieval client program GUI. One before the application integration and the other after the integration.

6.4 Related Work

In this section, we categorize 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 as follows.

First Category. 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. To provide transparency to provider or requester programs, approaches in this category typically use either the architecture illustrated in Figure 6.4 or in Figure 6.5, respectively. However, please note that the existence of translator components is not transparent to the programs hosting the translators. 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) [184], the Cal-Aggie Wrap-O-Matic project (CAWOM) [185], and the World Wide Web Factory (W4F) [186]. AIAG [184] 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. AIAG can be used to generate the required glue code to be used in client programs. CAWOM [185] 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 [186] is a Java toolkit that generates wrapper for Web resources. This toolkit provides a mapping mechanism for Java and XML.

Microsoft Visual Studio .NET [187] and IBM WebSphere Studio Application Developer [188] are among numerous commercial development environments that also fall in this category. Visual Studio .NET provides a set of visual tools enabling developers to integrate existing .NET programs with Web services. One example of such visual tools is the Add Web Reference GUI introduced in Section A.4 (see Figure A.19), which generates the proxy required to interoperate with Web services. Similarly, WebSphere provides a set of visual tools enabling a developer to transform existing components (*e.g.*, Java beans, EJB beans, and SQL statements) into Web services.

In addition to the development environments and visual tools, a number of command-line tools are also supported by Sun Microsystems and Microsoft to enable fast application integration by *generating* the required glue code. For example, Sun Microsystems Java Web Services Developer Pack (Java WSDP) [189], which is a free toolkit for developing Web services, provides command-line tools such as `wscompile.bat`, which can be used to generate stubs and skeletons from WSDL files. Microsoft also provides a set of command-line tools including `soapsuds.exe`, which generates a WSDL file from a .NET assembly file (and vice versa), and `wsdl.exe`, which generates code for Web service requester and provider programs from WSDL files, XSD schemas, and .discomap discovery documents. Examples of the use of these tools have been provided in Sections A.2, A.3, A.4, and 6.3.

Second Category. In the second category, we consider approaches that provide transparency with respect to both the provider and requester programs. Approaches in this category typically use the architectures illustrated in Figures 6.3, where a bridge program is used to host the translator components. Although such approaches provide transparency with respect to both requester and provider programs, they suffer from extra overhead imposed by one more level of process-to-process or machine-to-machine redirection. In addition, if only one bridge is used for integrating several programs, then the approaches in this category may also suffer from the single-point-of-failure and performance bottleneck problems. However, as described before, we note that localizing the translator components in one process has several advantages, among which the ease of maintenance is the most notable one.

Examples of standard specifications and commercial products in this category include SCOAP [190], CORBA Web services [191], Soap2Corba [171, 192], IONA Artix [193], and Apache Web Services Invocation Framework (WSIF) [194]. Typically, the goal in these approaches is to provide an *automatic* translation mechanism. In general, unless the automatic translation guarantees both syntax and semantic translations, such approaches may fail their purpose [173]. As an analogy, translating a conversation from English to French using a word-by-word translation approach, although may make sense in some few examples, but it generally fails the purpose. The reason behind such failure is that the semantics of application interactions are not captured only in the interface descriptions (*e.g.*, CORBA IDL and Web services WSDL). By translating just the syntax of CORBA IDL to WSDL for example, we cannot guarantee the semantic of heterogeneous applications are translated also.

For example, an object reference in CORBA cannot be mapped to a Web service address, because such addressing is not provided in Web services. In addition, consider exception and faults implemented in different ways in different middleware technologies cannot be easily translated to one another. Some proposals such as WS-Addressing [195], WSDL 2.0 [175], WS-Events [196], WS-Eventing [197], and WS-Notification [198] are examples of attempts to address these issues. Soap2Corba [192] provides a partial solution to representing a CORBA object references for Web services by translating object references to Web service "contexts" that are managed in a bridge between CORBA and web services applications. IONA Artix [193] and Apache Web Services Invocation Framework (WSIF) [194] provide a muti-middleware approach to application integration through multi-middleware routing and switching [173]. These two projects benefit from the WSDL abstractions and binding extensions and maximize the possibilities of integration.

Finally, an example of research projects in this category is the on-the-fly wrapping of Web services [199]. In this project, Web services are wrapped to be used by Java programs developed in Jini [200]. *Jini* is 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. Please note that a developer is required to complete the code for the bridge and to make sure 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.

Third Category. We consider transparent shaping in a third category. Similar to the approaches in the second category, transparent shaping provides transparency to provider and requester programs, and in addition, provides flexibility with respect to where the translator components are hosted. Transparent shaping provides alternative solutions to application integration as illustrated in Figures 6.3, 6.4, or 6.5. As discussed before, depending on the application needs and the integration requirements, one of these three architectures may be more appropriate. Transparent shaping benefits from the techniques provided in the first two categories and is considered as a complementary approach to the former approaches. We have shown how transparent shaping benefits from the techniques provided by the first category in Sections A.2, A.3, A.4, and 6.3. We plan to employ the automatic translation techniques provided by the approaches in the second category in our future work.

6.5 Summary

In this chapter, we have demonstrated how transparent shaping can be used to facilitate transparent application integration. Using TRAP/J and ACT/J, we provided alternative solutions to integrate heterogeneous applications. A case study was described, in which we used transparent shaping to integrate two existing applications, one of them was developed in CORBA and the other in .NET platform. Finally, we classified the approaches to application integration and discussed how transparent shaping relates to them. For our future work, we plan to extend the tools supporting transparent shaping and use the automatic translation techniques provided by other approaches to application integration. Specifically, for the former, our group is currently developing an implementation of TRAP for C++, and we plan to develop an implementation for C#. For the the latter, we plan to use Artix [193] as a supporting tool in transparent shaping.

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 ever increasing maturity of business standards, which have been supporting the automated interactions in business-to-business application integration over the past 20 years, addresses these issues to some extent [172]. Examples of some electronic businesses based on Web services include ebXML, RosettaNet, UCCNet, and XMethods. Also, the automatic service locating, which is one of the goals of Web services, has been specified in the Universal Description, Discovery, and Integration (UDDI) [201] specification. UDDI is a Web service for registering other Web services descriptions.

Chapter 7

Conclusions

Transparent shaping supports reuse of existing programs in new, dynamic environments even though the specific characteristics of such new environments were not anticipated during the original design of the programs. In particular, many popular programs, not designed to be adaptable, are being ported to dynamic environments. Transparent shaping enables dynamic adaptation in such programs and promotes coarse-grained reuse of software. In the rest of this chapter, we summarize our specific contributions and achievements, and discuss the future work.

7.1 Contributions

This dissertation produced four main contributions. We first summarize the contributions, and then discuss our achievements as a whole.

1. Assessment of language support in dynamic adaptation. In the first part of our study, we assessed how appropriate programming language constructs can facilitate the development of adaptable programs. We used Adaptive Java, which extends Java with behavioral reflection, to design a component called *MetaSocket*, whose behavior and structure can be adapted at run time in response to external stimuli (*e.g.,* wireless channel conditions). We evaluated the use of MetaSockets in a case study, where we provided dynamic adaptation in an audio streaming application. In this case study, the adaptation hooks were realized by the MetaSocket infrastructure and adaptive code was realized by filters in MetaSockets.

Although MetaSockets proved to be useful in supporting dynamic adaptation, our study of them revealed the following two issues. First, to incorporate a MetaSocket into an existing program, we need to modify the program source code directly, which is not desirable. Second, once the existing program is modified to use a MetaSocket instead of a Java socket, dynamic adaptation is only possible *within* the MetaSocket (*e.g.,* through the insertion and removal of filters). In other words, we cannot replace one version of a MetaSocket with another more appropriate version of the MetaSocket at run time.

2. Transparent, dynamic adaptation in object-oriented programs. In the second part of our study, we designed an extension of transparent shaping called Transparent Reflective Aspect Programming (TRAP), which supports dynamic adaptation in existing object-oriented programs *transparently*. A prototype of TRAP for Java, called *TRAP/J*, was developed and used to evaluate the TRAP concept. TRAP/J employs the structural reflection provided in Java and the aspect weaver provided in AspectJ [103] to support partial behavioral reflection [151] in existing Java programs. TRAP/J first generates wrapper and meta classes and weaves them into an existing program at compile time to generate an adapt-ready version of the program. Next, the adapt-ready program can be adapted at run time by insertion and removal of delegates.

In TRAP/J, a hook is realized by a pair of wrapper and meta classes associated with a class in the existing Java program, and adaptive code is realized by delegates, which can modify the behavior of the class by overriding the implementation of its methods. We developed a delegate using a MetaSocket, which in its turn supports dynamic adaptation through insertion and removal of filters. As a result, at run time, a MetaSocket can be replaced with a more appropriate one, if required. Results of this study showed the improvement in the execution of the audio streaming application in a mobile computing environment, while the adaptation is completely transparent to the application code.

3. Transparent, dynamic adaptation in CORBA programs. In the third part of our study, we designed and evaluated another extension of transparent shaping, called the Adaptive CORBA Template (ACT), which supports dynamic adaptation in existing CORBA programs transparently. ACT implements interception and redirection inside the supporting middleware instead of the program code itself. In addition, ACT enables interoperation among otherwise incompatible adaptive CORBA frameworks.

We developed an instance of ACT in Java, called *ACT/J*, to evaluate ACT in practice. In case studies, we showed the overhead introduced by ACT/J is negligible, while the adaptation provided is highly flexible. Specifically, we used ACT/J to enable an existing image retrieval program, originally designed for wired network, to continue working correctly in a mobile computing environment, where network may become disconnected and reconnected at any point in the execution of the program. In ACT, hooks are realized by a generic CORBA portable interceptor and adaptive code is realized by rule-based dynamic interceptors and their corresponding rules.

4. Transparent application integration. Finally, in the last part of our study, we assessed the potential role of transparent shaping beyond the scope of a single program. We demonstrated how transparent shaping can be used to support application integration. We proposed several alternative architectures and showed how transparent shaping can support interoperability, via Web service, for Java RMI, CORBA, and .NET applications. A case study demonstrated the use of transparent shaping in integration of an image retrieval application developed in CORBA and a frame grabber application developed in .NET.

7.2 Achievements

Figure 7.1 summarizes the achievements of this dissertation as a whole and shows how the contributions discussed above are related to one another. Transparent shaping is proposed as a new programming model, which supports dynamic adaptation in existing programs. Two instances of transparent shaping are provided: TRAP and ACT. TRAP employs a language-based approach and augments the existing programs with partial behavioral reflection using transformers at compile time. In contrast, ACT employs a middleware-based approach and augments the supporting middleware with a generic interceptor using CORBA portable interceptors. Prototype implementations of TRAP and ACT are provided in Java (TRAP/J and ACT/J, respectively).

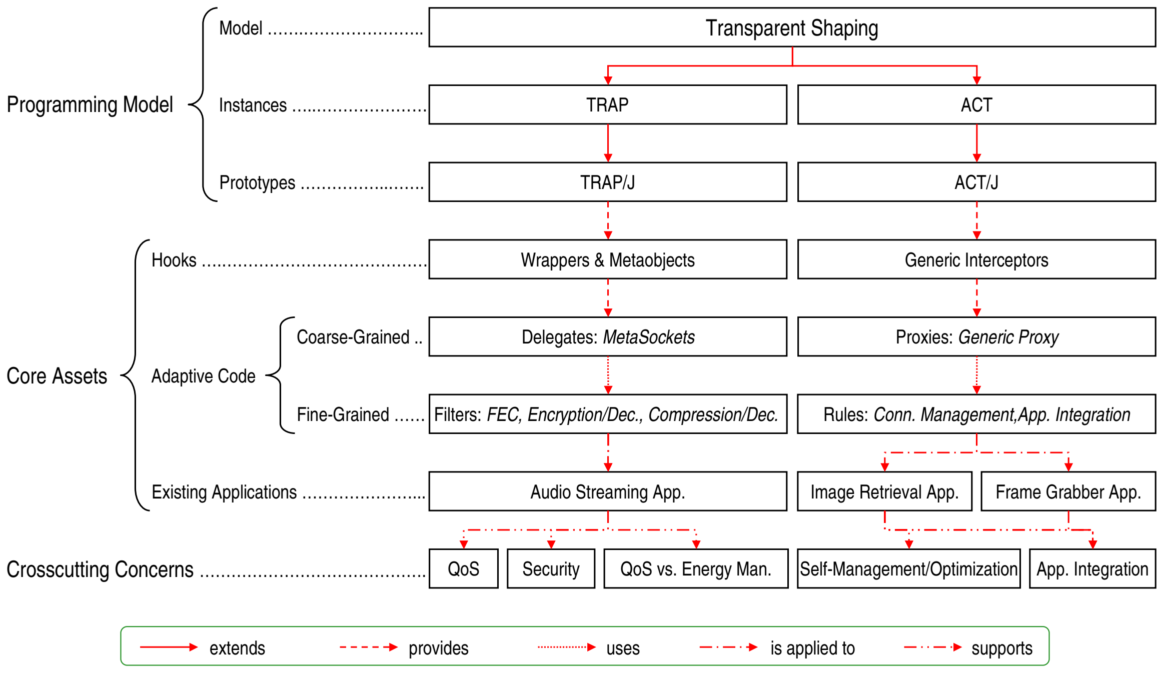


Figure 7.1: Achievements of this dissertation viewed as a whole.

TRAP/J and ACT/J are two concrete instances of transparent shaping, which can be used in the development of adaptable software several core assets supporting such product lines are developed. The core artifacts have been developed, including examples of hooks, adaptive code, and existing programs. The hooks in TRAP/J are pairs of wrappers and meta classes, which are generated by TRAP/J generators automatically. In ACT/J, there is only one hook, which is the generic portable interceptor. The generic portable interceptor was developed once and can be reused in any CORBA program. Adaptive code in TRAP/J is realized by developing delegates. A reusable delegate using MetaSockets and filters is provided. A generic proxy was developed for ACT/J that can be used in any existing CORBA applications. The generic proxy can receive any CORBA request and can adapt it using adaptive code realized by rules. Other reusable adaptive code (*e.g.,* filters, delegates, and rules) can be developed incrementally during the life time of a product line.

We have used TRAP/J and ACT/J to support dynamic adaptation in three existing applications, namely, an audio streaming application previously developed in our group [53], an image retrieval application developed by BBN [42], and a frame grabber application developed by NETMaster and distributed by Code Project [202]. We conducted several case studies using these applications, where we addressed a number of crosscutting concerns including QoS, security, energy consumption, self-management, and application integration. Our results have been well received by the community through publications in several conferences and workshops. We look forward to continuing these investigations and further contributing to the understanding of the important, yet still emerging, area of dynamic adaptation to support pervasive and autonomic computing.

7.3 Future Work

The work presented in this dissertation opens a door to several future research directions. In the rest of this chapter, we discuss five directions of future work, which complement this dissertation.

Expanding the set of supported existing programs. The set of existing programs supported by transparent shaping can be greatly expanded. Figure 7.2 shows the areas where we plan to work in near future. To support existing programs developed in C++ and C# programming languages, TRAP/C++ and TRAP/C# need to be developed. In fact, members of our group have already started implementing the TRAP/C++ using compile-time meta-object protocols supported by Open C++ [84], instead of a compile-time aspect weaver used in TRAP/J. Also, to support CORBA programs developed using C++ ORBs, we plan to develop ACT/C++.

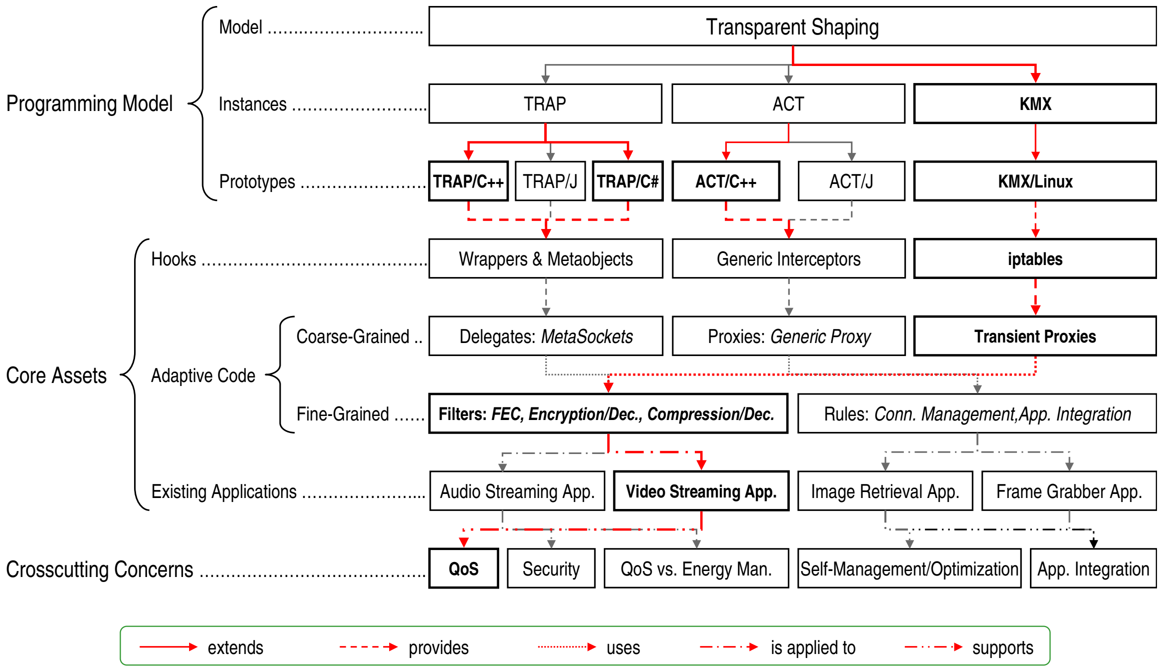


Figure 7.2: Expanding the set of existing programs supported by transparent shaping.

Transparent shaping can be instantiated using techniques other than those used in TRAP and ACT. In a related research project, called the Kernel Middleware eXchange (KMX), our group is investigating the use of iptables [203], which are a means of intercepting and redirecting network packets passing through an operating system kernel. A case study was conducted using a video streaming application previously developed in our group [204]. Originally, this application was developed as an adaptable program, which is capable of compensating the packet loss of a *one-hop* wireless network. In a preliminary study, our group adapted the flow of a video stream over a *multi-hop* wireless ad hoc network by intercepting and adapting the flow inside the intermediary nodes, which were used as routers. Using iptables as kernel-level hooks and transient proxies as adaptive code running inside the intermediate nodes, we showed how the flow of a video stream can be adapted inside the network completely transparent from the application code. This research project is still ongoing and we plan to implement a complete instance of transparent shaping to support dynamic adaptation of multimedia streams in existing distributed, multimedia programs.

Coordinating the behavior of adaptable programs. Designing distributed systems that can adapt to their environments requires not only adaptation of individual components, but coordinated adaptation across system layers and across platforms [6, 205–208]. Our work can be complemented by studies in the design of an adaptation coordinator to orchestrate the adaptation provided by adaptive components that address overlapping or even conflicting concerns (*e.g.*, preserving both QoS and energy in handheld devices). Different components are likely to have been developed by different parties, and the developer must be able to integrate separately-designed adaptive mechanisms such that they cooperate to meet the needs of the application. One problem is that many of the adaptive software solutions proposed for different layers have been developed independently, and even solutions within the same layer are often not compatible. As illustrated in Figure 7.3, tools and methods are needed to enable developers to integrate the operation of adaptive components across layers of a single system and among different systems.

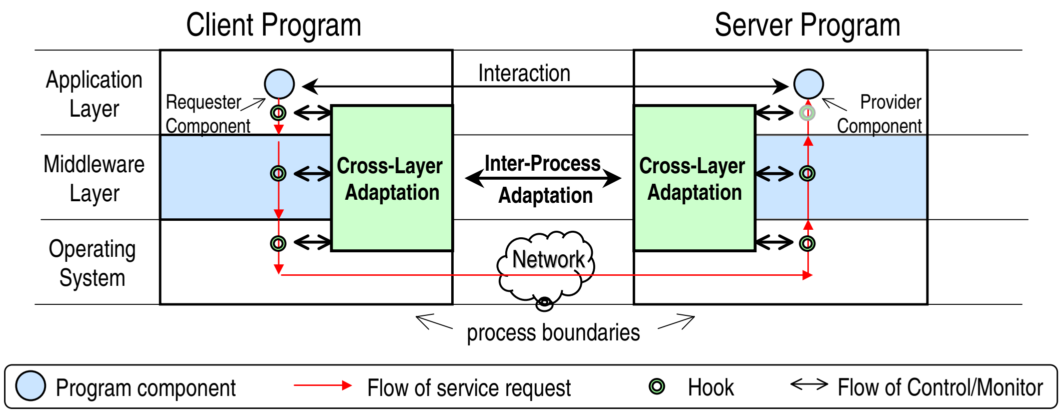


Figure 7.3: Future work on adaptation coordination.

Providing safe adaptation in adaptable programs. During the adaptation process, techniques are needed to ensure that the system continues to execute in an acceptable, or *safe* manner. Although transparent shaping enables dynamic insertion and removal of adaptive code, it does not guarantee a safe adaptation. For example, adaptive code may encapsulate part of the state of a running program and removing it without care may result in loss of state and unacceptable behavior by the running program. Our group and others are currently investigating this problem using a variety of methods, including dependency analysis [209–213] and explicit management of state information [214].

Providing security in adaptable programs. Security deals with protecting an adaptable program from malicious entities. An important issue is how to prevent the adaptation mechanisms from being exploited by a would-be attacker. In addition to verifying the sources of inserted components, the core of an adaptive software system must be extremely well protected from attackers. For example, the confidentiality and authenticity of messages related to adaptation must be ensured through strong encryption. The current prototypes of transparent shaping contain little support for preventing malicious entities from misusing the adaptation facilities. We plan to provide such support in the future versions of TRAP/J and ACT/J.

Constructing product lines for adaptable software. *Product line software engineering* [215–218] provides a disciplined methodology to produce program families. Transparent shaping can be used to construct software product lines specialized for "mass customization" [217] of existing programs to new environments. We plan to investigate design of *reactive* product lines (as opposed to proactive), where the product lines start from existing programs. The core assets of such product lines comprises existing programs, adaptation hooks, and adaptive code.

APPENDICES

Appendix A

Transparent Integration of Heterogeneous Distributed Applications

To show how transparent shaping and Web services can be used to integrate heterogeneous distributed applications, in this appendix we demonstrate transparent integration of applications developed in Java RMI [68], CORBA [47], and .NET [70] using a simple stock quote example. First, in Section A.1, we introduce a stock quote Web service. Next, in Sections A.2, A.3, and A.4, we use transparent shaping to integrate Web services with Java RMI, CORBA, and .NET applications, respectively. Specifically, for each of these three platforms, we show how a client program can be transparently shaped to use a Web Service (using architectures illustrated in Figures 6.6(a) and 6.6(b)) and how a server program can be exposed as a Web Service (using architectures illustrated in Figures 6.7(a) and 6.7(b)).[1] As discussed in Section 6.2, once two applications interoperate with Web services, then they are able to interoperate with each other via Web services.

A.1 A Stock Quote Web Service

Throughout this appendix, we use a stock quote Web service provided by XMethods (URL: http://www.xmethods.net/). This Web service, named `net.xmethods.services.-`

[1]Please note that the terms client and server programs are used equivalently with the terms requester and provider programs, respectively. Similarly, the terms client and server objects are equivalent to the terms requester and provider objects, and to the terms requester and provider components, respectively.

`stockquote.StockQuote`, provides stock quotes with 20 minutes of delay. The interface to this Web service is defined with the following method: `float getQuote(string symbol)`.

Figure A.1 shows a SOAP request representing the `getQuote(‘‘Microsoft")` method call. We note that this SOAP message does not have a header element. The method call is defined in the body element (lines 4 to 6). Similarly, Figure A.2 shows a SOAP response representing the `float result = -1`. The actual reply is defined in the body element (lines 4 to 6).

```
1     ≺?xml version=“1.0” encoding=“UTF-8” ?≻
2     ≺soap:Envelope xmlns:n=“urn:xmethods-delayed-quotes” . . . ≻
3       ≺soap:Body . . . ≻
4         ≺n:getQuote≻
5           ≺symbol xsi:type=“xs:string”≻Microsoft≺/symbol≻
6         ≺/n:getQuote≻
7       ≺/soap:Body≻
8     ≺/soap:Envelope≻
```

Figure A.1: A SOAP request example.

```
1   ≺?xml version=“1.0” encoding=“UTF-8” ?≻
2     ≺soap:Envelope xmlns:n=“urn:xmethods-delayed-quotes” . . . ≻
3       ≺soap:Body≻
4         ≺n:getQuoteResponse ≻
5           ≺Result xsi:type=“xsd:float”≻-1.0≺/Result≻
6         ≺/n:getQuoteResponse≻
7       ≺/soap:Body≻
8     ≺/soap:Envelope≻
```

Figure A.2: A SOAP response example.

Figure A.3 shows the corresponding WSDL of this Web service (available at `http://-services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl`). This WSDL describes an *abstract* application-level service description (interface) to the Web service (lines 4 to 16) as well as *concrete* protocol-dependent details of how to access the service (lines 18 to 39).

The abstract description part (lines 4 to 16) describes the interface to the Web service using the `message` elements (lines 4 to 9), which defines what type of messages can be sent to and received from the Web service, and the `portType` element (lines 11 to 16), which defines all the operations supported by the Web service. The `getQuote` operation (lines 12 to 15) defines the valid message exchange pattern supported by the Web service. The SOAP messages in Figures A.1 and A.2 are examples of the `input` and `output` messages,

```
 1   ≺?xml version=“1.0” encoding=“UTF-8” ?≻
 2   ≺definitions name=“net.xmethods.services.stockquote.StockQuote” . . . ≻
 3
 4     ≺message name=“getQuoteResponse1”≻
 5       ≺part name=“Result” type=“xsd:float” /≻
 6     ≺/message≻
 7     ≺message name=“getQuoteRequest1”≻
 8       ≺part name=“symbol” type=“xsd:string” /≻
 9     ≺/message≻
10
11     ≺portType name=“net.xmethods.services.stockquote.StockQuotePortType”≻
12       ≺operation name=“getQuote” parameterOrder=“symbol”≻
13         ≺input message=“tns:getQuoteRequest1” /≻
14         ≺output message=“tns:getQuoteResponse1” /≻
15       ≺/operation≻
16     ≺/portType≻
17
18     ≺binding name=“net.xmethods.services.stockquote.StockQuoteBinding”
19       type=“tns:net.xmethods.services.stockquote.StockQuotePortType”≻
20       ≺soap:binding style=“rpc” transport=“http://schemas.xmlsoap.org/soap/http” /≻
21       ≺operation name=“getQuote”≻
22         ≺soap:operation soapAction=“urn:xmethods-delayed-quotes#getQuote” /≻
23         ≺input≻
24           ≺soap:body use=“encoded” namespace=“urn:xmethods-delayed-quotes”
25             encodingStyle=“http://schemas.xmlsoap.org/soap/encoding/” /≻
26         ≺/input≻
27         ≺output≻
28           ≺soap:body use=“encoded” namespace=“urn:xmethods-delayed-quotes”
29             encodingStyle=“http://schemas.xmlsoap.org/soap/encoding/” /≻
30         ≺/output≻
31       ≺/operation≻
32     ≺/binding≻
33
34     ≺service name=“net.xmethods.services.stockquote.StockQuoteService”≻
35       ≺port name=“net.xmethods.services.stockquote.StockQuotePort”
36         binding=“tns:net.xmethods.services.stockquote.StockQuoteBinding”≻
37         ≺soap:address location=“http://targethost:9090/soap” /≻
38       ≺/port≻
39     ≺/service≻
40   ≺/definitions≻
```

Figure A.3: WSDL for net.xmethods.services.stockquote.StockQuote.

respectively, in the `operation` description.

The concrete description part (lines 18 to 39) complements the abstract part using the `binding` and `service` element. The `binding` element describes *how* a given interaction is performed over *what* specific transport protocol (lines 18 to 32). The `service` element describes *where* to access the service (lines 34 to 39). The *how* part describes how marshaling and unmarshaling is performed using the `operation` element inside the `binding` element (lines 21 to 31). The *what* part is described in line 20 using the `transport` attribute. The *where* part is described using the `port` element (lines 35 to 38).

A.2 Transparent Integration of Java RMI Applications and Web Services

Typically, a Java RMI application is composed of a client and server programs. Figure A.4 lists excerpted code for a client/server stock quote application developed in Java RMI. The code in this figure shows the contents of three files: the `StockQuoteInterface.java` file (lines 1 to 4) defines the interface to a stock quote remote object, the `StockQuoteServer.java` file (lines 5 to 20) defines the server program that hosts a stock quote remote object, and finally the `StockQuoteClient.java` file (lines 22 to 33) defines the client program that uses the stock quote service provided by the remote object. In the rest of this section, we show how the client and server programs can be transparently shaped to interoperate with Web service requester and provider programs, respectively.

A.2.1 Enabling Java RMI Client Applications to Use Web Services

In this part, we use the architectures illustrated in Figures 6.6(a) and 6.6(b) to enable the Java RMI client program to use the delayed stock quote Web service developed by XMethods (`URL: http://www.xmethods.net/`) introduced in Section 6.1.

First, let us discuss the architecture in Figure 6.6(b), where the Web service provider

```
 1  // The interface defined in StockQuoteInterface.java
 2  public interface StockQuoteInterface extends Remote {
 3    public float getQuote(String symbol) throws RemoteException;
 4  }
 5
 6  // The server application defined in StockQuoteServer.java
 7  public class StockQuoteServer extends UnicastRemoteObject {
 8    implements StockQuoteInterface { . . .
 9    public float getQuote(String symbol) throws RemoteException {
10      if (symbol.equalsIgnoreCase(“Microsoft”)) return 1;
11      else if (symbol.equalsIgnoreCase( “IBM”)) return 2;
12      else return -1;
13    }
14    public static void main(String args[]) {
15      try {
16        StockQuoteInterface obj = new StockQuoteServer();
17        Naming.rebind(“StockQuoteServer”, obj);
18      } catch (Exception e) {. . . }
19    }
20  }
21
22  // The client application defined in StockQuoteClient.java
23  public class StockQuoteClient {
24    public static void main(String[] args) {
25      try {
26        StockQuoteInterface stockQuote = (StockQuoteInterface)Naming.lookup(args[0]);
27        while (true) {
28          System.out.println(“Stock quote for IBM is: ” + stockQuote.getQuote(“IBM”));
29          Thread.sleep(1000);
30        }
31      } catch (Exception e) {. . . }
32    }
33  }
```

Figure A.4: A stock quote application developed in Java RMI.

is the XMethods Web server and the requester program is our Java RMI client application. We used the TRAP/J generator framework introduced in Chapter 4 to shape the Java RMI client program to host a translator, and for the translator to be able to intercept, translate, and forward all the Java RMI requests to the XMethods Web service.

Looking more closely to the Java RMI client program shown in Figure A.4 (lines 22 to 33), we can see that a reference to the remote object is obtained through a call to the `lookup()` static method of the `java.rmi.Naming` class (line 26). Using TRAP/J, we make the `java.rmi.Naming` class adaptable so that we can overwrite the implementation of its `lookup()` method transparently. Next, we provide a specific implementation for the `lookup()` method. Instead of its normal operation (that is contacting a Java RMI registry to find the reference to the remote object), it instantiates a translator object and returns a reference to the translator object to the client program.

Figures A.5 and A.6 show the excerpted code for the files generated by TRAP/J generators as a result of making the `java.rmi.Naming` class adaptable. The generated aspect is defined in the `Absorbing_Naming_aj.java` file (Figure A.5) that defines a `pointcut` to the calls to the `lookup()` method of the `java.rmi.Naming` class (lines 3 to 4). The `around` advice calls the `lookup()` static method of the wrapper-level class associated with the `Naming` class instead of the `lookup()` method of the `Naming` class (lines 5 to 8).

```
1   // Generated aspect defined in Absorbing_Naming_aj.java
2   public aspect Absorbing_Naming_aj {
3     pointcut lookup_String(String p0) :
4       call(public static Remote lookup(String)) && . . . ;
5     Remote around(String p0) throws . . . , RemoteException
6       : lookup_String(p0) {
7       return (Remote)WrapperLevel_Naming.lookup(p0);
8     }
9   }
```

Figure A.5: Excerpted code of the aspect generated by TRAP/J to shape the Java RMI client application.

The generated wrapper-level class is defined in the `WrapperLevel_Naming.java` file (Figure A.6, lines 1 to 19). It basically reifies calls to the `lookup()` method (lines 4 to 9) and forwards the reified method calls to the `invokeMetaMethod()` method of the static meta-class (`staticMetaClass`) associated with the wrapper class (lines 11 to 13). In case the reply to a method call is not provided by the `staticMetaClass` class, the `lookup()` method of the `java.rmi.Naming` class is called (lines 14 to 16). Finally, the reply is returned (line 17).

The generated meta-level class is defined in the `MetaLevel_Nameing.java` class (Figure A.6, lines 21 to 29). The static part of this class instantiates a meta-level class

```
 1  // Generated wrapper-level class defined in WrapperLevel_Naming.java
 2  public class WrapperLevel_Naming implements WrapperLevel_Interface { ...
 3    public static Remote lookup(String p0) throws ... {
 4      Class[] paramType = new Class[1]; paramType[0] = String.class;
 5      Method method = null;
 6      try { method = WrapperLevel_Naming.class.getMethod("lookup", paramType); }
 7      catch (Exception e) {...}
 8      Object[] tempArgs = new Object[1]; tempArgs[0] = p0;
 9      ChangeableBoolean isReplyReady = new ChangeableBoolean(false);
10      Remote retVar = null;
11      try { retVar = (Remote) MetaLevel_Naming.staticMetaClass.
12        invokeMetaMethod(method, tempArgs, isReplyReady);
13      } catch (Exception e) {...}
14      if(!isReplyReady.booleanValue()) {
15        retVar = (Remote)Naming.lookup(p0);
16      }
17      return retVar;
18    } ...
19  }
20
21  // Generated meta-level class defined in MetaLevel_Nameing.java
22  public class MetaLevel_Naming extends UnicastRemoteObject
23    implements MetaLevel_Interface, DelegateManagement {
24    public synchronized Object invokeMetaMethod(
25      Method method,Object[] args,ChangeableBoolean isReplyReady) throws ... { ... }
26    static public MetaLevel_Naming staticMetaClass;
27    static { try { staticMetaClass = new MetaLevel_Naming();} catch (Exception e) {} }
28    private MetaLevel_Naming() throws RemoteException { ... }
29  }
```

Figure A.6: Excerpted code of the reflective classes generated by TRAP/J to shape the Java RMI client application.

(`staticMetaClass`) associated with the wrapper-level class using its private constructor (lines 26 to 28). This class provides the `invokeMetaMethod()` method that forward reified method calls to delegates that overwrite the corresponding method implementation. Compiling the generated code listed in Figures A.5 and A.6 together with the Java RMI client program using the AspectJ [103] compiler, an adapt-ready version of the client program is created. This adapt-ready program is adapted at startup time using a configuration file. The configuration file instructs the adapt-ready program to insert a delegate to the `staticMetaClass`. The delegate provides the new implementation for the `lookup()` method.

Figure A.7 shows excerpted code of the delegate class defined in the `Delegate_Naming_lookup.java` file (lines 1 to 6) and the translator component defined in the `StockQuoteProxy.java` file (lines 8 to 25). The `Delegate_Naming_lookup` class simply provides a new implementation for the `lookup()` method that return an instance of the `StockQuoteProxy` instead of using the Java RMI registry to find a reference to the Java RMI remote object (lines 3 to 5). The `StockQuoteProxy` class provides an implementation of the translator component that for the Java RMI client program plays the role of a Java RMI remote object and for the XMethods delayed stock quote Web service plays the role of a requester object. A reference to the delayed stock quote Web service (`stockQuoteWebService`) is obtained using the WSDL available at `URL: http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl` (listed in Figure A.3) and the Java Web Services Developer Pack (Java WSDP) version 1.4 (lines 11 to 20). The implementation of the `getQuote()` method (lines 21 to 24) simply returns the result from the call to the `getQuote()` of the XMethods stock quote Web service.

Because the stock quote example is very simple and because the XMethods stock quote Web service provides a similar service as that of our Java RMI remote object, the code for translation is very simple. As can be seen from the excerpted code for the `StockQuoteProxy` class, the developer did not need to provide any semantic translation. However, the code for translation, no matter how complicated, can be encapsulated in the `StockQuoteProxy` class. A more complicated example of application integration is provided in Section 6.3.

To make this application integration a bit more interesting, we listed an alternative code for the translator class in Figure A.8. Basically, this implementation provides a fault-tolerance service for the Java RMI client program. It keeps a reference to the original Java RMI remote object (`origStockQuoteServer` line 4 and lines 8 to 11) as well as a reference to the XMethods stock quote Web service (`stockQuoteWebService` line 6

```
 1  // The delegate defined in Delegate_Naming_lookup.java
 2  public class Delegate_Naming_lookup implements Delegate_Interface {
 3    public static Remote lookup(String p0, ChangeableBooleanisReplyReady) throws . . . {
 4      return new StockQuoteProxy(p0);
 5    }
 6  }
 7
 8  // The proxy defined in StockQuoteProxy.java
 9  public class StockQuoteProxy extends UnicastRemoteObject
10    implements StockQuoteInterface { . . .
11    private NetXmethodsServicesStockquoteStockQuotePortType stockQuoteWebService;
12    public StockQuoteProxy(String p0) throws RemoteException { super();
13      try {
14        Stub stub = (Stub)(new NetXmethodsServicesStockquoteStockQuoteService_Impl().
15          getNetXmethodsServicesStockquoteStockQuotePort());
16        stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
17          "http://64.124.140.30:9090/soap");
18        stockQuoteWebService=(NetXmethodsServicesStockquoteStockQuotePortType)stub;
19      } catch (Exception ex) {. . . }
20    }
21    public float getQuote(String symbol) throws RemoteException {
22      try { return stockQuoteWebService.getQuote(symbol); }
23      catch (Exception e) {. . . }
24    }
25  }
```

Figure A.7: Excerpted code of the delegate and proxy classes.

and lines 12 to 18). The `getQuote()` implementation is now fault-tolerant (lines 20 to 30). By default, the translator object first tries the original Java RMI remote object (line 22). In case this remote object fails to respond (*e.g.*, the network connection is temporarily disconnected, or the remote object, or for some reason it has crashed), the XMethods Web service is used (line 24). The translator component prefers to use the Java RMI remote object for the next calls (lines 25 to 27), because the Java RMI remote object imposes less delay than the XMethods Web service.

Alternatively, we can integrate the Java RMI client program with the XMethods delayed stock quote Web service using the other approach illustrated in Figure 6.6(a), where the translator component is hosted inside a bridge program. The code for the proxy component would be the same as the proxy class listed in Figures A.7 or A.8, except that it would have a `main()` method that instantiates and registers the proxy class as that of the Java RMI

```
// The proxy defined in StockQuoteProxy.java
public class StockQuoteProxy extends UnicastRemoteObject
  implements StockQuoteInterface { . . .
  private StockQuoteInterface origStockQuoteServer;
  private String StockQuoteServerName;
  private NetXmethodsServicesStockquoteStockQuotePortType stockQuoteWebService;
  public StockQuoteProxy(String p0) throws RemoteException {
    super(); StockQuoteServerName = p0;
    try { origStockQuoteServer = (StockQuoteInterface)
      Naming.lookup(StockQuoteServerName);
    } catch (Exception e) {. . . }
    try {
      Stub stub = (Stub)(new NetXmethodsServicesStockquoteStockQuoteService_Impl().
        getNetXmethodsServicesStockquoteStockQuotePort());
      stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
        ”http://64.124.140.30:9090/soap”);
      stockQuoteWebService=(NetXmethodsServicesStockquoteStockQuotePortType)stub;
    } catch (Exception ex) {. . . }
  }
  public float getQuote(String symbol) throws RemoteException {
    float retVal = -1;
    try { retVal = origStockQuoteServer.getQuote(symbol); }
    catch (Exception e) {
      retVal = stockQuoteWebService.getQuote(symbol);
      try { origStockQuoteServer = (StockQuoteInterface)
        Naming.lookup(StockQuoteServerName);
      } catch (Exception e1) {}
    }
    return retVal;
  }
```

Figure A.8: Excerpted code of the proxy class that adds fault-tolerance to the Java RMI client application by using a Web Service if the Java RMI server is not available.

server program (similar to the code in Figure A.4 lines 14 to 19). For brevity, we do not list the code for this approach.

A.2.2 Exposing Java RMI Server Applications as Web Services

In this part, we show how the Java RMI server program (listed in Figure A.4, lines 6 to 20) can be exposed as a Web service using the architectures illustrated in Figures 6.7(a) and 6.7(b).

First, let us discuss the architecture in Figure 6.7(a), where we use a Web server as a server-side bridge program to host the translator component. For the Web server, we use the

Tomcat Web server from Apache Software Foundation redistributed with the Java WSDP version 1.4 (`URL: http://java.sun.com/webservices/jwsdp/index.jsp`).

Figure A.9 lists the excerpted code for the translator component that we developed and deployed on a Tomcat Web server. The interface is defined in the `StockQuoteWSInterface.java` file (lines 1 to 4) and the Web service is defined in the `StockQuoteWS.java` file (lines 6 to 18). Basically, the Web service obtains a reference to the Java RMI remote object (lines 8 to 13) and implements the `getQuote()` method by forwarding the calls to the Java RMI remote object (lines 14 to 17).

```
 1   // The Web service interface defined in StockQuoteWSInterface.java
 2   public interface StockQuoteWSInterface extends Remote {
 3     public float getQuote(String symbol) throws RemoteException;
 4   }
 5
 6   // The Web service implementation defined in StockQuoteWS.java
 7   public class StockQuoteWS implements StockQuoteWSInterface {
 8     static StockQuoteInterface stockQuoteRMIServer;
 9     static {
10       try { stockQuoteRMIServer = (StockQuoteInterface)
11         Naming.lookup("//localhost/StockQuoteServer"); }
12       catch (Exception e) { . . . }
13     }
14     public float getQuote(String symbol) {
15       try {return stockQuoteRMIServer.getQuote(symbol);}
16       catch (Exception e) { return -1;}
17     }
18   }
```

Figure A.9: Excerpted code of the Web service.

Alternatively, as illustrated in Figure 6.7(b), we can host the translator program inside the Java RMI server program. For this approach, we can use TRAP/J to overwrite the implementation of the `rebind()` method of the `java.rmi.Naming` class. When the `rebind()` method is called (Figure A.4 line 17), the new implementation of the `rebind()` method (provided in a delegate class) that instantiates a proxy object. The proxy object keeps a reference to the original Java RMI remote object and exposes itself as a Web service. When a requester program sends a message to the Web service (the proxy object), it simply forwards the request to the original Java RMI remote object, which is a local object for the proxy object, and replies to the `getQuote()` SOAP messages using the result form the Java RMI object.

A.3 Transparent Integration of CORBA Applications and Web Services

Similar to Java RMI applications, a typical CORBA application is composed of a client and server programs. Figures A.10, A.11, and A.12 list the excerpted code for a simple implementation of the stock quote application in CORBA: the stock quote interface is defined in the `StockQuoteInterface.idl` file, the server code is defined in the `StockQuoteServer.java` file, and the client code is defined in the `StockQuoteClient.java` file. In the rest of this section, we show how the client and server programs were transparently shaped to interoperate with Web service requester and provider programs.

```
// The stock quote interface defined in StockQuoteInterface.idl
module edu { module msu { module cse { module sens { module StockQuote {
  interface StockQuoteInterface { float getQuote(in string symbol); }
}; }; }; }; };
```

Figure A.10: The stock quote interface defined in IDL.

A.3.1 Enabling CORBA Client Applications to Use Web Services

We use the architectures illustrated in Figures 6.6(a) and 6.6(b) to enable the CORBA client application to use the delayed Quote Web service.

First, we use the architecture in Figure 6.6(b), where the translator component is hosted inside the client program, to make the CORBA client program interoperate with the XMethods delayed stock quote Web service. Before describing the code listed in Figures A.13 and A.14, let us take a closer look at the client program listed in Figure A.12. First, a

```
// The stock quote server implemented in CORBA defined in StockQuoteServer.java
public class StockQuoteServer extends StockQuoteInterfacePOA {
  public float getQuote(String symbol) { ... }
  public static void main(String[] args) {
    try {
      ORB orb = ORB.init( args, null );
      POA poa = POAHelper.narrow( orb.resolve_initial_references( "RootPOA" ));
      poa.the_POAManager().activate();
      StockQuoteServer server = new StockQuoteServer();
      org.omg.CORBA.Object obj = poa.servant_to_reference( server );
      PrintWriter pw = new PrintWriter( new FileWriter( args[ 0 ] ));
      pw.println( orb.object_to_string( obj )); pw.flush(); pw.close();
      orb.run();
    } catch( Exception e ) {...}
  }
}
```

Figure A.11: Implementation of the stock quote server program in CORBA.

reference to the CORBA object (`server` defined in line 3) is obtained in the constructor of the `StockQuoteClient` class using the IOR file provided to the client program as a command line parameter (lines 5 to 12). Next, this reference is used to call the `getQuote()` method on the CORBA object hosted in the server program (lines 16 to 18).

To intercept and redirect the CORBA requests, first, we make the client program adapt-ready by running the program using extra command-line parameters: `java StockQuoteClient` *`ORBconfig file:StockQuoteClient.cfg`*.[2] Next, the developer inserts a new rule to the rule-based decision maker of the ACT core that intercepts all the CORBA requests using GUI tools.

Figures A.13 and A.14 list the excerpted code that we developed to make the CORBA program interoperate with the XMethods Web service. The condition part of the rule is defined in the `StockQuote_Condtion.java` file (Figure A.13, lines 1 to 8), which basically returns `true` always to make all the intercepted CORBA requests to be forwarded to the action part of the rule. The action part of the rule is defined in the `StockQuote_Action.java` file (lines 10 to 25). In the constructor of the (`Stock-`

[2]For details of how ACT works, please refer to Chapter 5.

```
// The stock quote client implemented in CORBA defined in StockQuoteClient.java
public class StockQuoteClient {
  StockQuoteInterface server;
  public StockQuoteClient(String args[]) {
    try {
      ORB orb = ORB.init( args, null );
      String iorFileName = args[0]; File f = new File( iorFileName );
      BufferedReader br = new BufferedReader( new FileReader( f ));
      org.omg.CORBA.Object obj = orb.string_to_object( br.readLine() );
      br.close();
      server = StockQuoteInterfaceHelper.narrow( obj );
    } catch( Exception ex ) {...}
  }
  public static void main( String args[] ) {
    StockQuoteClient client = new StockQuoteClient(args);
    try {
      while (true) { System.out.println(...+ client.server.getQuote("IBM")); ... }
    } catch( Exception ex ) {...}
  }
}
```

Figure A.12: Implementation of the stock quote client program in CORBA.

`Quote_Action`) class (Figure A.13, lines 12 to 17), an instance of the translator component (defined in the `StockQuote_ClientLocalProxy.java` file listed in Figure A.14) is created.

Once the rule is inserted, all CORBA requests will be reified by the CORBA ORB and will eventually be intercepted by the `process()` method of the `StockQuote_Action` class (lines 18 to 25). The `process()` method creates another CORBA request similar to the one intercepted, except that the target object of the request is `stockQuote_Client-LocalProxy`, which is an instance of the `StockQuote_ClientLocalProxy` class, instead of the original CORBA object. The `process()` method replies the original request using the reply returned from the `stockQuote_ClientLocalProxy`.

The translator component is defined in the `StockQuote_ClientLocalProxy.java` file. First, a reference to the XMethods Web service is obtained using the Java WSDP framework (Figure A.14, lines 5 to 11). Next, all calls to the `getQuote()` method are forwarded to the XMethods Web service (lines 14 to 15).

```
// The condition class defined in StockQuote_Condtion.java
public class StockQuote_Condition extends InBandDMCondition {
  public StockQuote_Condition(ActiveRule activeRule, ORB orb) { super(activeRule, orb); }
  public boolean check(org.omg.CORBA.Object targetObj, FullInterfaceDescription
    fullIntDesc, ServerRequest serverRequest, Request request) {
    return true;
  }
}

// The action class defined in StockQuote_Action.java
public class StockQuote_Action extends edu.msu.cse.sens.act.dm.InBandDMAction {
  public StockQuote_Action(ActiveRule activeRule, org.omg.CORBA.ORB orb) {
    super(activeRule, orb);
    StockQuote_ClientLocalProxy stockQuote_ClientLocalProxy =
      new StockQuote_ClientLocalProxy(orb);
    // publishing the StockQuote_ClientLocalProxy CORBA object in the naming service
  }
public boolean process(org.omg.CORBA.Object targetObj, FullInterfaceDescription
  fullIntDesc, ServerRequest serverRequest, Request request) {
  request = createReq(stockQuote_ClientLocalProxy, serverRequest, fullIntDesc, opNum);
  request.invoke();
  org.omg.CORBA.Any res_any = request.result().value();
  serverRequest.set_result(res_any);
  return true;
}
```

Figure A.13: Excerpted code of the rule used in the client program.

A.3.2 Exposing CORBA Server Applications as Web Services

We propose three solutions to expose CORBA server applications as Web services. The first two solutions use the architecture in Figure 6.7(b) and the last one uses the architecture in Figure 6.7(a). First, we can use TRAP/J to host the translator component inside the CORBA server application. This approach is possible only if the CORBA application is written in Java. However, when other implementations of TRAP, such as TRAP/C++ and TRAP/C#, are available, we can use this approach for CORBA applications written in C++ and C#, respectively. Second, we can use ACT to host the translator component inside the CORBA server application. Third, we can use the architecture illustrated in Figure 6.7(a), where a server-side bridge hosts the translator component. Because those solutions involve straightforward modifications to code described earlier, we do not list the corresponding

```
// The stock quote proxy defined in StockQuote_ClientLocalProxy.java
public class StockQuote_ClientLocalProxy extends StockQuotePOA
  implements Serializable, StockQuoteOperations {
  public StockQuote_ClientLocalProxy(ORB orb) { ...
    try {
      Stub stub = (Stub)(new NetXmethodsServicesStockquoteStockQuoteService_Impl().
        getNetXmethodsServicesStockquoteStockQuotePort());
      stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
        "http://64.124.140.30:9090/soap");
      stockQuoteWebService=(NetXmethodsServicesStockquoteStockQuotePortType)stub;
    } catch (Exception ex) {...}
  }
  public float getQuote(String symbol) throws RemoteException {
    try { return stockQuoteWebService.getQuote(symbol); }
    catch (Exception e) {...}
  }
}
```

Figure A.14: Excerpted code of the proxy used in the client program.

code.

A.4 Transparent Integration of .NET Remoting Applications and Web Services

Microsoft .NET [70] is a collection of software technologies intended to support integration of small, discrete applications as well as larger applications over the Internet. Similar to Java RMI and CORBA applications, a typical .NET remoting application is composed of client and server programs. In .NET remoting, unlike CORBA there is no need to define the interface of a CORBA object in an IDL file, and unlike Java RMI there is no need to define the interface of a remote object separately from its implementation.

Figures A.15, A.16, and A.17 list excerpted code for a client/server stock quote application developed in C#. The `StockQuoteObject.cs` file defines both the interface and implementation of the remote objects of type `StockQuoteObject`. The `StockQuoteServer.cs` file defines the server program that follows the instructions listed in its configuration file. The `StockQuoteServer.exe.config` file defines the configuration of the server program. It instructs the server program to define its name as "StockQuote-

Server” (Figure A.16, line 10) and to listen to the port number 9000 that may receive `http` requests (line 15). If a SOAP message is received that refers to the `StockQuoteObject` service (lines 11 to 14), then the server program instantiates an object of type `StockQuote.StockQuoteObject` (available in the `StockQuoteObject.dll` library file) for the message and forwards the message to the object.[3]

```
// The stock quote object defined in StockQuoteObject.cs
public class StockQuoteObject : MarshalByRefObject {
  public float getQuote(string symbol) {
    if (symbol.Equals(“Microsoft”)) return 1; . . .
  }
}
```

Figure A.15: Excerpted code of the stock quote object in the .NET remoting application.

```
 1   // The stock quote server defined in StockQuoteServer.cs
 2   class StockQuoteServer {
 3     [STAThread] static void Main(string[] args) {
 4       RemotingConfiguration.Configure(“StockQuoteServer.exe.config”);
 5       String keyState = “”; keyState = Console.ReadLine();
 6     }
 7   }
 8
 9   // The stock quote server configuration defined in StockQuoteServer.exe.config
10   ≺configuration≻ ≺system.runtime.remoting≻ ≺application name=“StockQuoteServer”≻
11     ≺service≻
12       ≺wellknown mode=“SingleCall” type=“StockQuote.StockQuoteObject,
13         StockQuoteObject” objectUri=“StockQuoteObject.soap” /≻
14       ≺/service≻
15     ≺channels≻ ≺channel port=“9000” ref=“http” /≻ ≺/channels≻
16   ≺/application≻ ≺/system.runtime.remoting≻ ≺/configuration≻
```

Figure A.16: Excerpted code of the server program in the .NET remoting application.

The `StockQuoteClient.cs` file (listed in Figure A.17) defines the client program. The client program first follows the configuration provided in its configuration file (line 4). Next, it creates an instance of the `StockQuoteObject` class (line 5). Finally, it uses

[3]Please note that the `mode="SingleCall"` attribute in the configuration file (Figure A.16, line 12) instructs the server program to instantiate a separate object for each arriving message. Alternatively, if the `mode="Singleton"` attribute is used, only one singleton object will be created for all arriving messages.

```
 1  // The stock quote client defined in StockQuoteClinet.cs
 2  class StockQuoteClient {
 3    [STAThread] static void Main(string[] args) {
 4      RemotingConfiguration.Configure("StockQuoteClient.exe.config");
 5      StockQuoteObject server = new StockQuoteObject();
 6      while (true) {
 7        Console.WriteLine("Stock quote for IBM is: " + server.getQuote("IBM"));
 8        Thread.Sleep(1000);
 9      }
10    }
11  }
12
13  // The stock quote client configuration defined in StockQuoteClient.exe.config
14  ≺configuration≻ ≺system.runtime.remoting≻ ≺application name="StockQuoteClient"≻
15    ≺client≻
16      ≺wellknown type="StockQuote.StockQuoteObject, StockQuoteObject"
17        url="http://haydn.cse.msu.edu:9000/StockQuoteServer/StockQuoteObject.soap" /≻
18    ≺/client≻
19  ≺/application≻ ≺/system.runtime.remoting≻ ≺/configuration≻
```

Figure A.17: Excerpted code of the client program in the .NET remoting program.

the .NET remote object (lines 6 to 9). The `StockQuoteClient.exe.config` file (lines 13 to 19) defines the configuration of the client program. This configuration file instructs the client program to define its name as "StockQuoteClient" (line 14) and to register the `StockQuote.StockQuoteObject` type as a remote object. The effect of this registration is that whenever the client program instantiates a new object of type `StockQuote.-StockQuoteObject` (such as line 5), the instance is not created locally. Instead, a proxy is automatically created that represents the .NET remote object as a local object. Whenever a call to the object is made (such as line 7), the call is forwarded to the service residing at the "`http://haydn.cse.msu.edu:9000/StockQuoteServer/StockQuote-Object.soap`" URL (line 17). Eventually, the call will be received by the server program, where an instance of type `StockQuote.StockQuoteObject` is created to respond to the request.

A.4.1 Enabling .NET Client Applications to use Web Services

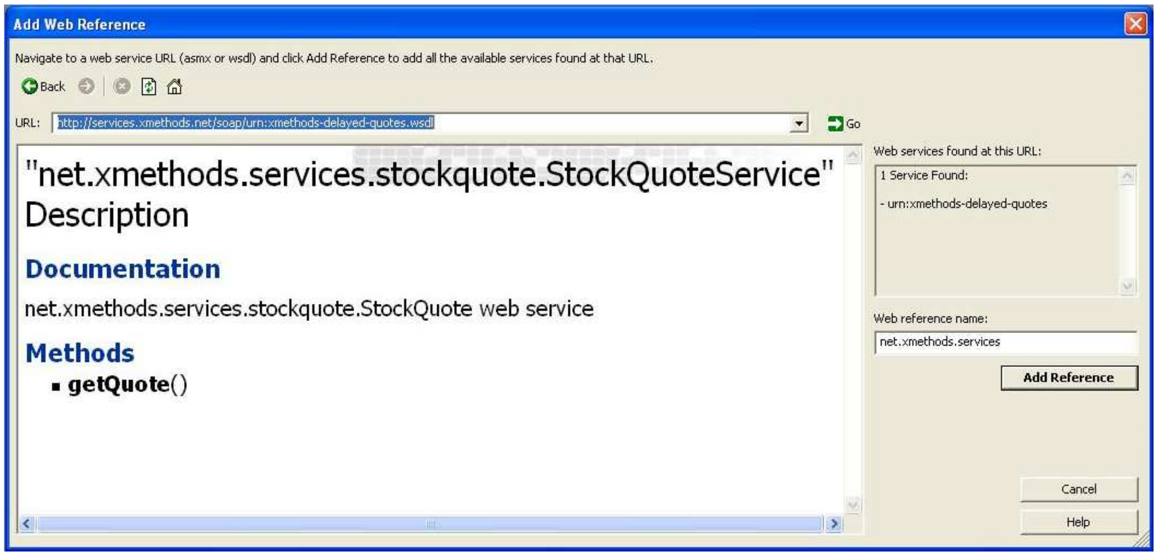
In this part, we show how the architectures illustrated in Figures 6.6(a) and 6.6(b) can be used to enable the .NET client program to use the XMethods Web service.

First, we use the architecture in Figure 6.6(a), where a bridge program hosts the translator component. Figure A.18 lists the excerpted code for the translator component defined in the `StockQutoeObject.cs` file. First, a reference to the XMethods Web service (`webService`) is obtained (lines 4 to 6). Next, calls to the `getQuote()` method are forwarded to the `webService` (lines 7 to 9). The code for the bridge program that hosts this translator component is the same as the one for the server program listed in Figure A.16.

```
 1  // The stock quote proxy defined in StockQutoeObject.cs
 2  public class StockQuoteObject : MarshalByRefObject {
 3    private netxmethodsservicesstockquoteStockQuoteService webService;
 4    public StockQuoteObject() {
 5      webService = new netxmethodsservicesstockquoteStockQuoteService();
 6    }
 7    public float getQuote(string symbol) {
 8      return webService.getQuote(symbol);
 9    }
10  }
```

Figure A.18: Excerpted code of the translator component that is hosted inside a bridge program.

The `netxmethodsservicesstockquoteStockQuoteService` type, representing the XMethods Web service, is available to the translator program by creating the corresponding proxy class using either the `wsdl.exe` utility from Microsoft or the `Add Web Reference` facility in the `VisualStudio .NET`. All the plumbing and marshaling are hidden in the generated Web service proxy and the translator program simply uses the `netxmethodsservicesstockquoteStockQuoteService` type. Figure A.19 shows a screen dump of the `Add Web Reference` GUI that we used to make available the web reference to the `http://services.xmethods.net/soap/urn:-xmethods-delayed-quotes.wsdl` WSDL definition for the XMethods Web service.


Add Web Reference

Navigate to a web service URL (asmx or wsdl) and click Add Reference to add all the available services found at that URL.

URL: http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl

"net.xmethods.services.stockquote.StockQuoteService" Description

Documentation

net.xmethods.services.stockquote.StockQuote web service

Methods

- getQuote()

Web services found at this URL:

1 Service Found:

- urn:xmethods-delayed-quotes

Web reference name: net.xmethods.services

Figure A.19: A screen dump of the Add Web Service GUI of VisualStudio .NET that adds the reference to the XMethods Web service to a .NET project.

Alternatively, we could use the architecture illustrated in Figure 6.6(b), where the translator component is hosted inside the client program. However, because this example is developed in C# and we do not have the C# version of TRAP yet, currently the alternative solution is not available.

A.4.2 Exposing .NET Server Applications as Web Services

A .NET server can be used directly as a Web service, if the types exposed by the .NET server are all supported in XML schema type system[4]. The types that can be used in a .NET remoting application are much richer than the restricted types supported in Web services [182, 183]. If a .NET specific type such as `System.Data.DataSet` is exposed by a .NET server program, then it cannot be used as a Web service directly. As our .NET server program does not expose such .NET specific types, it can be used directly as a Web service. To enable a Web requester program to use this service, a developer can use the `SOAPsuds.exe` utility from Microsoft with the `-sdl` option to generate the WSDL schema of the .NET service. This WSDL schema can be used by Web service requester

[4]The XML schema type system specifies all the types that are allowed to be used in a Web service.

programs to interoperate with the .NET server program. If a .NET specific type is used in a .NET server program, then the corresponding WSDL schema will have some extra elements, which are not standard and may not be understood by other Web services tools.

In case a .NET server cannot be used directly as a Web service, we can expose it as a Web service using the architecture illustrated in Figure 6.7(b), where the translator component is hosted inside the provider program. We use the IIS Web server from Microsoft (`http://www.microsoft.com/WindowsServer2003/iis/default.mspx`) distributed with Windows XP as the provider program. We use the ASP.NET technology to host the provider component inside the IIS Web server (for this example the provider component is an instance of the `StockQuoteObject` class listed in Figure A.18). The excerpted code for the Web service is listed in Figure A.20. First, the Web service creates an instance of the `StockQuoteObject` and stores it in the `origServer` variable (lines 3 to 6). Next, it forwards calls to the `getQuote()` method to the `origServer`.

```
 1   // The translator program defined in StockQuoteWebService.cs.
 2   public class StockQuoteWebService : System.Web.Services.WebService {
 3     StockQuoteObject origServer;
 4     public StockQuoteWebService() {
 5       origServer = new StockQuoteObject();
 6     }
 7     [WebMethod (EnableSession=true)]
 8     public float getQuote(string symbol) {
 9       origServer.getQuote(symbol);
10     }
11   }
```

Figure A.20: Excerpted code for the Web service written in ASP.NET and hosted inside the IIS Web server.

Alternatively, we can expose the .NET server program as a Web service using the architecture illustrated in Figure 6.7(a), where the translator component is hosted inside a bridge program. For this approach, we use the IIS Web server as the bridge program. This approach is similar to the first solution discussed above (see Figure A.20), except that the object can be instantiated as a .NET remote object instead of a local object.

BIBLIOGRAPHY

Bibliography

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, June 1997.

[2] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6), June 2002.

[3] Overview of ACE. Available at URL: `http://www.cs.wustl.edu/~schmidt/ACE-overview.html`.

[4] Overview of the ACE+TAO Project. Available at URL: `http://www.cs.wustl.edu/~schmidt/TAO-overview.html`.

[5] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[6] M. Roman, F. Kon, and R. H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[7] Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha 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.

[8] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In *Proceedings of the International Conference on Congurable Distributed Systems (ICCDS'98)*, May 1998.

[9] R. Koster. *A Middleware Platform for Information Flows*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, July 2002.

[10] Overview of CORBA. Available at URL: `http://www.cs.wustl.edu/~schmidt/corba-overview.html`.

[11] M. Weiser. Ubiquitous computing. *IEEE Computer*, 26(10):71–72, October 1993.

[12] João Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the third Working IEEE/IFIP Conference on Software Architecture*, pages 29–43, 2002.

[13] The Oxygen project at MIT. Available at URL: `http://oxygen.lcs.mit.edu/`.

[14] The Planet Blue project at IBM. Available at URL: `http://www.research.ibm.com/compsci/planetblue.html`.

[15] M. Roman and R. Campbell. Gaia: Enabling active spaces. In *Proceedings of the ninth ACM SIGOPS European Workshop*, Kolding, Denmark, 2000.

[16] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[17] *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004.

[18] Paul Horn. Manifesto on autonomic computing, 2001. Available at URL: `http://www.research.ibm.com/autonomic`.

[19] *IBM Systems Journal, Special issue on Autonomic Computing*, volume 42, 2003.

[20] E. W. Dijkstra. Structured programming. *Software Engineering Techniques, edited by Buxton and Randell (available from NATO, Brussels)*, pages 84–87, 1970.

[21] Daniel M. Hoffman and David M. Weiss. *Software fundamentals: collected papers by David L. Parnas*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[22] Peri Tarr and Harold Ossher, editors. *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (W17)*, May 2001.

[23] Karl Lieberherr. *Adaptive Object-Oriented Software The Demeter Method*. PWS Publishing Company, Boston, 1996.

[24] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming*. Addison Wesley, 2000.

[25] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *International Conference on Software Engineering*, pages 120–130, 1999.

[26] *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, volume 44, October 2001.

[27] Pierre Charles David, Thomas Ledoux, and Noury M. N. Bouraqadi-Saadani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, October 2001.

[28] Martin Geier, Martin Steckermeier, Ulrich Becker, Franz J. Hauck, Erich Meier, and Uwe Rastofer. Support for mobility and replication in the AspectIX architecture. Technical Report TR-I4-98-05, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

[29] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *Proceedings of the International Conference on Software Engineering*, pages 174–186, Portland, Oregon, May 2003.

[30] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, pages 147–155. ACM Press, December 1987.

[31] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocols*. MIT Press, 1991.

[32] A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous AOP with PROSE. Technical report, Department of Computer Science, Federal Institute of Technology, Zurich, 2001.

[33] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[34] David E. Bakken. *Middleware*. Kluwer Academic Press, 2001.

[35] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on The future of Software engineering*, pages 117–129, 2000.

[36] Andrew T. Campbell, Geoff Coulson, and Michael E. Kounavis. Managing complexity: Middleware explained. *IT Professional, IEEE Computer Society*, (5):22–28, September/October 1999.

[37] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy 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.

[38] G. S. Blair, G. Coulson, P. Robin, and 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.

[39] Andrew T. Campbell and Michael E. Kounavis. Toward reflcetive network architecture. In *Middleware'2000 Workshop on Reflective Middleware (RM2000)*, New York, April 2000.

[40] T. Ledoux and N. Bouraqadi-Saadani. Adaptability in mobile agent systems using reflection. In *Middleware'2000 Workshop on Reflective Middleware (RM2000)*, New York, April 2000.

[41] Mark Astley, Daniel C. Sturman, and Gul A. Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, 2001.

[42] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

[43] Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco, and Jehan Wickramasuriya. Design and implementation of a composable reflective middleware. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001.

[44] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, April 1998.

[45] Raymond Klefstad, Douglas C. Schmidt, and Carlos 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.

[46] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, March 1976.

[47] Object Management Group, Framingham, Massachusett. *The Common Object Request Broker: Architecture and Specification Version 3.0*, July 2003. Available at URL: `http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf`.

[48] Geoff A. Cohen, Jeffrey S. Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 Usenix Technical Conference*, June 1998.

[49] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. E. K. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, pages 465–472, Vienna, Austria, July 2002.

[50] P. K. McKinley, S. M. Sadjadi, and E. P. Kasten. An adaptive software approach to intrusion detection and response. In *Proceedings of The 10th International Conference on Telecommunication Systems, Modeling and Analysis (ICTSM10)*, pages 91–99, Monterey, California, October 2002.

[51] P. K. McKinley, E. P. Kasten, S. M. Sadjadi, and Z. Zhou. Realizing multi-dimensional software adaptation. In *Proceedings of the ACM Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, held in conjunction with the *16th Annual ACM International Conference on Supercomputing*, New York City, NY, June 2002.

[52] Philip K. McKinley, S. M. Sadjadi, E. P. Kasten, and R. Kalaskar. Programming language support for adaptive wearable computing. In *Proceedings of International Symposium on Wearable Computers (ISWC'02)*, pages 205–214, Seattle, Washington, October 2002.

[53] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the Ninth IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 46–55, San Juan, Puerto Rico, May 2003.

[54] Z. Zhou, P. K. McKinley, and S. M. Sadjadi. On quality-of-service and energy consumption tradeoffs in fec-enabled audio streaming. In *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS 2004)*, Montreal, Canada, June 2004. Winner of the IWQoS 2004 best student paper award.

[55] S. M. Sadjadi, P. K. McKinley, R. E. K. Stirewalt, and B. H.C. Cheng. Generation of self-optimizing wireless network applications. In *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*, pages 310–311, New York, NY, May 2004.

[56] S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E. Kurt 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. To appear.

[57] 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.

[58] 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)*, pages 88–95, New York, NY, May 2004.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[60] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[61] E. S. Hudders. *CICS: A Guide to Internal Structure.* Wiley, 1994.

[62] C. Lo. Hall. *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.

[63] Lo Gilman and R. Schreiber. *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[64] M. Hapner, R. Burridge, and R. Sharma. Java message service specification. Technical report, Sun Microsystems, Nov. 1999.

[65] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *ACM Transactions on Computer Systems 2(1)*, pages 39–59, March 1984.

[66] Open Group. *DCE 1.1: Remote Procedure Calls*, 1997.

[67] Object Management Group, Framingham, Massachusett. *The Common Object Request Broker: Architecture and Specification Revision 2.2*, February 1998.

[68] Java Soft. *Java Remote Method Invocation Specification, revision 1.5, JDK 1.2*, Oct. 1998.

[69] Microsoft Corporation. *Microsoft COM Technologies - DCOM*, 2000.

[70] Microsoft, Available at URL: `http://www.microsoft.com/net/`. *Microsoft .NET*.

[71] Microsolft Corporation. *COM: Delivering on the Promises of Component Technology*, 2000. Available at URL: `http://www.microsoft.com/com/default.asp`.

[72] David Conger. *Remoting with C# and .NET*. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[73] Gopalan Suresh Raj. A detailed comparison of CORBA, DCOM, and Java/RMI (with detailed code examples). *Object Management Group (OMG) whitepaper*, Sept. 1998.

[74] Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang. DCOM and CORBA side by side. Technical report, Microsoft DCOM Technical White Paper, 1997.

[75] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture*, volume 2. John Wiley, 2001.

[76] R. Rao. Implementational reflection in Silica. In *Proceedings of ECOOP'91*, pages 251–267. Springer-Verlag, 1991.

[77] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13, Jan. 1996.

[78] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of ACM*, (10):51–57, October 2001.

[79] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran. Evaluating meta-programming mechanisms for ORB middleware. *IEEE Communications Magazine, Special Issue on Evolving Communications Software: Techniques and Technologies*, October 2000.

[80] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[81] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA'93*, 1993.

[82] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, Jan 1982.

[83] Gordon Blair, Geoff Coulson, and Nigel 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 (NOSSDAV)*, pages 259–273, 1997.

[84] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. *Lecture Notes in Computer Science*, 707, 1993.

[85] J. McAffer. Meta-level architectue support for distributed obejcts. In *Proceedings of Reflection'96*, pages 39–62, San Francisco, California, 1996.

[86] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 127–144, New York, NY, 1992. ACM Press.

[87] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the Workshop on New Models for Software Architecture*, Nov. 1992.

[88] Yasuhiko Yokote. Kernel structuring for object-oriented operating systems: The Apertos approach. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 145–162. Springer-Verlag, 1993.

[89] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: an operating systems perspective. Technical Report UIUCDCS–R–92–1736, University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana-Champaign, 1992.

[90] D. B. Orr. Applications of meta-protocols to improve OS services. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 101–105, May 1995.

[91] R. Hayton. *FlexiNet Open ORB Framework*. APM Ltd., Oct. 1997.

[92] Thomas Ledoux. OpenCorba: A reflective open broker. *Lecture Notes in Computer Science*, 1616, 1999.

[93] M. Roman, M. Mickunas, F. Kon, and R. H. Campbell. LegORB and ubiquitous CORBA. In *Proc. IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM2000)*, New York, April 2000.

[94] Jean Charles Fabre and Tanguy Perennou. A mebject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[95] W. Cazzola and M. Ancona. mChaRM: A reflective middleware for communications-based reflection. Technical Report DISI-TR-00-09, Universita degli Studi di Milano, May 2000.

[96] Michael Golm. Design and implementation of a meta architecture for Java. Master's thesis, Friedrich-Alexander-University, Erlangen-Nurenburg, Jan. 1997.

[97] Vikram Adve, Vinh Vi Lam, and Brian Ensink. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, Snowbird, Utah, June 2001.

[98] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, June 2002.

[99] Nanbor Wang, Douglas C. Schmidt, and Michael Kircher. Towards an adaptive and reflective middleware framework for QoS-enabled CORBA component model applications. *IEEE Distributed System Online, Special Issue on Reflective Middleware*, 2003.

[100] A. Singhai, A. Sane, and R. H. Campbell. Quarterware for middleware. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 192–201, Amesterdam, The Netherlands, May 1998.

[101] Sun Microsystems. *Enterprise JavaBeans Technology*, 2001. Available at URL: `http://java.sun.com/products/ejb/`.

[102] Object Management Group. *CORBA Components Model - FTF drafts for MOF cahpter*. Available at URL: `http://www.omg.org/cgi-bin/doc?ptc/99-10-05`.

[103] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[104] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[105] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[106] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient solution for aspect-oriented programming in Java. In *Proceedings of Reflection 2001, LNCS 2192*, pages 1–24, September 2001.

[107] Ian Welch and Robert J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 157–169. Springer-Verlag, Heidelberg, Germany, June 2000.

[108] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communication of the ACM*, October 2001.

[109] Z. Yang, B. H.C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02)*, November 2002.

[110] Michel Cukier, Jennifer Ren, Chetan Sabnis, David Henke, Jessica Pistole, William H. Sanders, David E. Bakken, Mark E. Berman, David A. Karr, and Richard E. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 245. IEEE Computer Society, Oct. 1998.

[111] Christian R. Becker and Kurt Geihs. MAQS: Management for adaptive QoS-enabled services. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, 1997.

[112] A. Corsaro, D. Schmidt, R. Klefstad, and 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.

[113] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[114] Ossama Othman. The design, optimization, and performance of an adaptive middleware load balancing service. Master's thesis, University of California, Irvine, 2002.

[115] D. Sharp. Reducing avionics software cost through component-based product line development. In *Proceedings of the Software Technology Conference*, Salt Lake City, Utah, April 1998.

[116] R. Baldoni, C. Marchetti, and A. Termini. Active software replication through a three-tier approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages 109–118, Osaka, Japan, October 2002.

[117] Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies*, pages 135–146, 1995.

[118] Sun Microsystems. *EmbeddedJava Application Environment*. Available at URL: `http://java.sun.com/products/embeddedjava/`.

[119] IONA Technologies Inc. *ORBacus for C++ and Java version 4.1.0*, 2001.

[120] Gerald Brose and Nicolas Noffke. JacORB 1.4 documentation. Technical report, Freie Universitt Berlin and Xtradyne Technologies AG, August 2002.

[121] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and 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.

[122] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, pages 95–106, September 2002.

[123] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2003.

[124] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2002.

[125] Kelvin Nilsen. Issues in the design and implementation of real-time java. *Java Developers Journal*, 1996.

[126] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware resource management for distributed multimedia applications. *Special issue on multimedia networking, J. High Speed Network*, Dec. 1998.

[127] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Thread transparency in information flow middleware. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer Verlag, November 2001.

[128] G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, H. A. Duran, R. Moreira, N. Paralavantzas, and K. B. Saikoski. The design and implementation of open ORB version 2. 2(6), 2001.

[129] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. *Lecture Notes in Computer Science*, 2218, 2001.

[130] ITU-T/ISO. *Reference Model for Open Distributed Processing, Parts 1,2,3.*, 1995. ITU-T X.901-X.904 — ISO/IEC IS 10746-(1,3,3).

[131] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applicaitons (OPSLA'88)*, pages 306–315, San Diego, California, Sept. 1988.

[132] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the International Workshop on Multimedia Middleware*, pages 44–47. ACM, October 2001.

[133] Jie Huang, Andrew Black, Jonathan Walpole, and Calton Pu. Infopipes - an abstraction for information flow. In *Proceedings of the ECOOP Workshop on The Next 700 Distributed Object Systems*, Budapest, Hungary, 2001.

[134] P. K. McKinley, U. I. Padmanabhan, N. Ancha, and S. M. Sadjadi. Composable proxy services to support collaboration on the mobile internet. *IEEE Transactions on Computers (Special Issue on Wireless Internet)*, pages 713–726, June 2003.

[135] S. E. Hudson, editor. *CUP User's Manual*. Usability Center, Georgia Institute of Technology, july 1999.

[136] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[137] L. Rizzo and L. Vicisano. RMDP: An FEC-based reliable multicast protocol for wireless environments. *ACM Mobile Computer and Communication Review*, 2(2), April 1998.

[138] David A. Eckhardt and Peter Steenkiste. A trace-based evaluation of adaptive error correction for a wireless local area network. *Mobile Networks and Applications*, 4(4):273–287, 1999.

[139] J. Andersson and T. Ritzau. Dynamic code update in JDrums. In *Proceedings of the ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, 2000.

[140] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, Universidade Estadual de Campinas, September 1998.

[141] José de Oliveira Guimarães. Reflection for statically typed languages. In *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, pages 440–461, 1998.

[142] Eddy Truyen, Bo N. Jörgensen, Wouter Joosen, and Pierre Verbaeten. Aspects for run-time component integration. In *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia Antipolis and Cannes, France, 2000.

[143] F. Akkai, A. Bader, and T. Elrad. Dynamic weaving for building reconfigurable software systems. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida, October 2001.

[144] D. Wagelaar. Towards a context-driven development framework for ambient intelligence. In *Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, Tokyo, Japan, March 2004.

[145] S. Ren, M. Beckman, and T. Elrad. System imposed and application compliant adaptation. In *Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, Tokyo, Japan, March 2004.

[146] R. Hirschfeld and K. Kawamura. Dynamic service adaptation. In *Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, Tokyo, Japan, March 2004.

[147] Shigeru Chiba. Load-time structural reflection in Java. *Lecture Notes in Computer Science*, 1850, 2000.

[148] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proceedings of OORaSE*, pages 117–133, 1999.

[149] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the first International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002.

[150] Denis Caromel and Julien Vayssière. Reflections on MOPs, Components, and Java Security. In J. Lindskov Knudsen, editor, *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 256–274, Budapest, Hungary, June 2001. Springer-Verlag.

[151] Éric Tanter, Jacques Noyè, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, 2003. ACM Press.

[152] Sun Microsystems. *Java 2 Platform, Standard Edition*, 2001. Available at URL: `http://www.sun.com/j2se/`.

[153] M. Golm and J. Kleinoder. metaXa and the future of reflection. In *Proceedings of Workshop on Reflective Programming in C++ and Java*, pages 1–5, 1998.

[154] Zhixue Wu. Reflective Java and a reflective component-based transaction architecture. In *Proceedings of Workshop on Reflective Programming in C++ and Java*, 1998.

[155] Eduardo Kessler Piveta and Luiz Carlos Zancanella. Aspect weaving strategies. *Journal of Universal Computer Science*, 9(8):970–983, 2003.

[156] Erez Hadad. *Architectures for Fault-Tolerant Object-Oriented Middleware Services*. PhD thesis, Computer Science Department, The Technion - Israel Institute of Technology, 2001.

[157] Orbix+Isis programmer's guide. Technical Report D071-00, ISIS DISTRIBUTED SYSTEMS, INC., IONA TECHNOLOGIES, LTD., 1995.

[158] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software Practice and Experience*, 28(9):963979, August 1998.

[159] R. Friedman and 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.

[160] Sadaf Mumtaz and Naveed Ahmad. Architecture of kaffe. Available at URL: `http://wiki.cs.uiuc.edu/cs427/Kaffe+Architecture+Project+Site`.

[161] I. Welch and R. Stroud. Dalang — a reflective extension for java. Technical Report CS-TR-672, University of Newcastle upon Tyne, East Lansing, Michigan, September 1999.

[162] P. Felber, B. Garbinato, and R. Guerraoui. Towards reliable CORBA: Integration vs. service approach. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 199–205. dpunkt-Verlag, 1997.

[163] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[164] S. M. Sadjadi and P. K. McKinley. Supporting transparent and generic adaptation in pervasive computing environments. Technical Report MSU-CSE-03-32, Department of Computer Science, Michigan State University, East Lansing, Michigan, November 2003.

[165] John Zinky, Joseph Loyall, and Richard 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.

[166] H.-Arno Jacobsen and Bernd Kraemer. A design pattern based approach for generating synchronization adaptors from annotated IDL. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 63–72, 1998.

[167] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed system. Technical Report 422, Vrije Universiteit, Amsterdam, The Netherlands, March 1997.

[168] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault. The Totem system. In *Proceedings of the 25th International Symposium on Fault Tolerant Computing*, pages 61–66, Pasadena, California, 1995.

[169] Mads Haahr, Raymond Cunningham, and Vinny Cahill. Supporting CORBA applications in a mobile environment. In *Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking*, 1999.

[170] Object Management Group. *Common Object Services Specification*. John Wiley & Sons, Inc., 1994. Available at URL: `http://www.omg.org/` for the latest specification.

[171] Aniruddha Gokhale, Bharat Kumar, and Arnaud Sahuguet. Reinventing the wheel? CORBA vs. Web services. In *Proceedings of International World Wide Web Conference*, Honolulu, Hawaii, 2002.

[172] Steve Vinoski. Where is middleware? *IEEE Internet Computing*, March-April 2002.

[173] Steve Vinoski. Integration with Web services. *IEEE Internet Computing*, November-December 2003.

[174] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. *Web Services Architecture*. W3C, 2004. Available at URL: `http://www.w3.org/TR/ws-arch/`.

[175] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 edition, March 2004. Available at URL: `http://www.w3.org/TR/wsdl20/`.

[176] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *SOAP Version 1.2*. W3C, 1.2 edition, 2003. Available at URL: `http://www.w3.org/TR/soap12`.

[177] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1*. W3C, 1.1 edition, 2000. Available at URL: `http://www.w3c.org/TR/SOAP`.

[178] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[179] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 1.1 edition, March 2001. Available at URL: `http://www.w3c.org/TR/wsdl`.

[180] Gartner. *Application Integration & Web Services Summit 2004*, May 2004.

[181] Mark Pesce. *Programming Microsoft DirectShow for Digital Video*. Microsoft Press, 2003.

[182] Thiru Thangarathinam. .NET remoting versus Web services. *Online article*, 2003. Available at URL: `http://www.developer.com/net/net/article.php/11087_2201701_1`.

[183] Priya Dhawan and Tim Ewald. Building distributed applications with microsoft .net (ASP.NET Web services or .NET remoting: How to choose). *Online article*, September 2002. Available at URL: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/%html/bdadotnetarch16.asp`.

[184] Ngom Cheng, Valdis Berzins, Luqi, and Swapan Bhattacharya. Interoperability with distributed objects through java wrapper. In *Proceedings of the 24th Annual International Computer Software and Applications Conference,* Taipei, Taiwan, October 2000.

[185] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Generating wrappers for command line programs: the Cal-Aggie Wrap-O-Matic project. In *Proceedings of the 23rd international conference on Software engineering*, pages 243–252. IEEE Computer Society, 2001.

[186] Arnaud Sahuguet and Fabien Azavant. Looking at the Web through XML glasses. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*, pages 148–159, September 1999.

[187] Product Information for Visual Studio .NET 2003. Available at URL: `http://msdn.microsoft.com/vstudio/productinfo/`.

[188] C. Lau and A. Ryman. Developing xml web services with websphere studio application developer. *IBM Systems Journal,* July 2002.

[189] Java Web Services Developer Pack (Java WSDP). Available at URL: `http://java.sun.com/webservices/jwsdp/index.jsp`.

[190] BEA Systems Incorporation and et al. CORBA Web services. OMG TC Document orbos/2001-06-07. Available at URL: `http://soap2corba.sourceforge.net/`.

[191] Borland Software Corporation and et al. CORBA Web services. OMG TC Document orbos/2001-06-07. Available at URL: `http://soap2corba.sourceforge.net/`.

[192] Walter Stroebel. SOAP to CORBA bridge documentation, version 2.0.1. Available at URL: `http://soap2corba.sourceforge.net/`.

[193] IONA Technologies. *Artix: Tutorial Version 2.0.2*, 2003. Available at URL: `http://www.iona.com/support/docs/artix/2.0/tutorial/index.html`.

[194] Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. *Web Services Invocation Framework (WSIF)*. IBM T.J. Watson Research Center, August 2001. Available at URL: `http://ws.apache.org/wsif/`.

[195] Adam Bosworth, Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Tony Storey, and Sanjiva Weerawarana. *Web Services Addressing (WS-Addressing)*. BEA Systems, Microsoft, and IBM, March 2004. URL: ftp://www6.software.ibm.com/software/developer/library/ws-add200403.pdf.

[196] Nicolas Catania, Pankaj Kumar, Bryan Murray, Homayoun Pourhedari, William Vambenepe, and Klaus Wurster. *Web Services Events (WS-Events) Version 2.0*. Hewlett- Packard Company, July 2003. Available at URL: `http://devresource.hp.com/drc/specifications/wsmf/WS-Events.pdf`.

[197] Luis Felipe Cabrera, Craig Critchley, Gopal Kakivaya, Brad Lovering, Matt MihicDavid Orchard, Shivajee Samdarshi, Jeffrey Schlimmer, John Shewchuk, and David Wortendyke. *Web Services Eventing (WS-Eventing)*. Microsoft, BEA Systems, and TIBCO Software, January 2004. Available at URL: `http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf`.

[198] Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Steve Tuecke, William Vambenepe, and Bill Weihl. *Web Services Notification (WS-Notification) Version 1.0*. IBM, Sonic Software, TIBCO Software, Akamai Technologies, SAP AG, Globus / Argonne National Laboratory, Hewlett-Packard, January 2004. Available at URL: `http://ifr.sap.com/ws-notification/ws-notification.pdf`.

[199] Gerald C. Gannod, Huimin Zhu, and Sudhakiran 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.

[200] W. Keith Edwards. *Core Jini*. Prentice-Hall, 1999.

[201] Bob Atkinson, Tom Bellwood, Maud Cahuzac, Luc Clment, John Colgrave, Ugo Corda, Alexandru Czimbor, Matthew J. Dovey, Daniel Feygin, Shishir Garg, Rajul Gupta, Andrew Hately, Brad Henry, Aikichi Kawai, Paul Macias, Anne Thomas Manes, Claus von Riegen, Tony Rogers, Alok Srivastava, Paul Thorpe, Alessandro Triglia, Max Voskob, and George Zagelow. *UDDI Version 3.0.1*. OASIS, 2003. Available at URL: `http://uddi.org/pubs/uddi_v3.htm`.

[202] Code Project. Available at URL: `http://www.codeproject.com`.

[203] The netfilter/iptables project. Available at URL: `http://www.iptables.org/`.

[204] F. A. Samimi, P. K. McKinley, S. M. Sadjadi, and P. Ge. Kernel-middleware interaction to support adaptation in pervasive computing environments. Technical Report MSU-CSE-04-30, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, August 2004.

[205] Jun He, Matti A. Hiltunen, Mohan Rajagopalan, and Richard D. Schlichting. QoS customization in distributed object systems. *Software Practice and Experience*, 33(4):295–320, 2003.

[206] Distributed extensible open systems (the DEOS project), 2004. Georgia Institute of Technology - College of Computing.

[207] S. Adve, A. Harris, C. Hughes, D. Jones, R. Kravets, K. Nahrstedt, D. Sachs, R. Sasanka, J. Srinivasan, and W. Yuan. The illinois grace project: Global resource adaptation through cooperation, 2002.

[208] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-box Systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001. Available at URL: `http://www.cs.wisc.edu/graybox/`.

[209] Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, May 2004.

[210] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma M. Da Silva, Orran Krieger, David J. Edelsohn Marc A. Auslander, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal*, 42(1), 2003.

[211] Nalini Venkatasubramanian. Safe composability of middleware services. *Communications of the ACM*, 45(6), June 2002.

[212] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In *International Symposium on Component-based Software Engineering (CBSE7)*, May 2004.

[213] Noriki Amano and Takuo Watanabe. A software model for flexible and safe adaptation of mobile code programs. In *Proceedings of the international workshop on Principles of software evolution*, pages 57–61. ACM Press, 2002.

[214] E. P. Kasten and P. K. McKinley. Perimorph: Run-time composition and state management for adaptive systems. In *Proceedings of Fourth International Workshop on Distributed Auto-Adaptive and Reconfigurable Systems (DARES), in conjunction with ICDCS 2004*, pages 332–337, Hachioji, Japan, March 2004.

[215] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[216] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[217] The Software Engineering Institute (SEI). *A Framework for Software Product Line Practice (Version 4.2)*, 2004. Available at URL: `http://www.sei.cmu.edu/plp/framework.html`.

[218] The Software Engineering Institute (SEI). *Software Product Line Acquisition: A Companion to A Framework for Software Product Line Practice (Version 2.0)*, 2003. Available at URL: `http://www.sei.cmu.edu/plp/companion.html`.