TRAP/J: Transparent Generation of Adaptable Java Programs

S. Masoud Sadjadi[1], Philip K. McKinley[2], Betty H.C. Cheng[2], and
R.E. Kurt Stirewalt[2]

[1] School of Computer Science, Florida International University, Miami, FL 33199
`sadjadi@cs.fiu.edu`[* * *]
[2] Software Engineering and Network Systems Laboratory, Department of Computer
Science and Engineering, Michigan State University, East Lansing, Michigan 48824
`{mckinley,chengb,stire}@cse.msu.edu`

Abstract. This paper describes *TRAP/J*, a software tool that enables
new adaptable behavior to be added to existing Java applications trans-
parently (that is, without modifying the application source code and
without extending the JVM). The generation process combines behav-
ioral reflection and aspect-oriented programming to achieve this goal.
Specifically, TRAP/J enables the developer to select, at compile time,
a subset of classes in the existing program that are to be adaptable at
run time. TRAP/J then generates specific aspects and reflective classes
associated with the selected classes, producing an *adapt-ready* program.
As the program executes, new behavior can be introduced via interfaces
to the adaptable classes. A case study is presented in which TRAP/J
is used to introduce adaptive behavior to an existing audio-streaming
application, enabling it to operate effectively in a lossy wireless network
by detecting and responding to changing network conditions.

Keywords: generator framework, transparent adaptation, dynamic re-
configuration, aspect-oriented programming, behavioral reflection, mid-
dleware, mobile computing, quality-of-service.

1 Introduction

As the computing and communication infrastructure continues to expand and
diversify, developing software that can respond to changing conditions is be-
coming increasingly important. A notable example is the advent of the "Mobile
Internet," where software on handheld and wearable computing devices must
adapt to several potentially conflicting concerns, such as quality-of-service, se-
curity and energy consumption. Unfortunately, many distributed applications
being ported to mobile computing environments were not designed to adapt to
changing conditions involving such concerns. We say that a program is *adaptable*
if it contains facilities for selecting and incorporating new behavior at run time.

[* * *] This research was performed while this author was a graduate student at Michigan
State University.

Adaptable applications can be difficult to develop and maintain. In particular, adaptive code associated with concerns such as quality of service tends to crosscut the conventional "functional decomposition" of an application. Therefore, manually attempting to modify an existing program to support adaptation can be tedious and error-prone, and usually produces adaptive code that is tangled throughout the functional code of the application.

In this work, we investigate how to transparently enhance existing applications so that they can be adapted to cross-cutting concerns at run time. By *transparent*, we mean that the new behavior is added without modifying the application source code and without extending the JVM. This study is part of RAPIDware, an ONR-sponsored project that addresses the design of adaptable software for network-centric battlefield environments and for protecting critical infrastructures from component failures and cyber-attack. However, we also expect the resulting software technologies to apply to more general mobile Internet applications, where transparency is increasingly important.

The predominant mechanism for implementing adaptation in object-oriented software is *behavioral reflection* [1,2], which can be used to modify how an object responds to a message. In recent years, behavioral reflection has been used to support adaptation to a variety of different concerns, including quality of service and fault tolerance. Unfortunately, programs that use behavioral reflection typically incur additional overhead, since in some cases every message sent to an object must be intercepted and possibly redirected. To improve efficiency, a developer should be able to *selectively* introduce behavioral reflection only where needed to support the desired adaptations.

In earlier work [3], we 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 the existing program.

This paper describes transparent reflective aspect programming (TRAP), which combines behavioral reflection [1] and aspect-oriented programming [4] to transform extant programs into *adapt-ready* programs automatically and transparently. TRAP supports general behavioral reflection by automatically generating wrapper classes and meta-classes from selected classes in an application. TRAP then generates aspects that replace instantiations of selected classes with instantiations of their corresponding wrapper classes. This two-pronged, automated approach enables selective behavioral reflection with minimal execution overhead. To validate these ideas, we developed TRAP/J, which instantiates TRAP for Java programs. In an earlier poster summary [5], we discussed the *use* of TRAP/J in wireless network applications. In this paper, we focus on the *operation* of TRAP/J and describe the details of the techniques used to generate adapt-ready programs and their reconfiguration at run time. TRAP/J source code is available for download at the RAPIDware homepage (www.cse.msu.edu/rapidware).

The remainder of this paper is organized as follows. Section 2 presents background information, categorizes research projects that address adaptability in distributed applications, and discusses how TRAP relates to other approaches. Section 3 describes the operation of the TRAP/J prototype. Section 4 presents a case study in which we used TRAP/J to augment an existing audio-streaming application with adaptive behavior, enabling it to operate more effectively across wireless networks. Finally, Section 5 summarizes the paper and discusses future investigations.

2   Background and Related Work

Many approaches to constructing adaptable software, including TRAP/J, use behavioral reflection, aspect-oriented programming, or a combination of both. In this section, we briefly review these technologies in the context of Java, followed by a discussion of projects related to TRAP/J.

2.1   Behavioral Reflection and Java

According to Maes [1], *behavioral* or *computational* reflection 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 parts unnecessarily [2]. As depicted in Figure 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. A *meta-object protocol (MOP)* is a set of meta-level interfaces that enables "systematic" (as opposed to ad hoc) inspection and modification of the base-level objects [2].
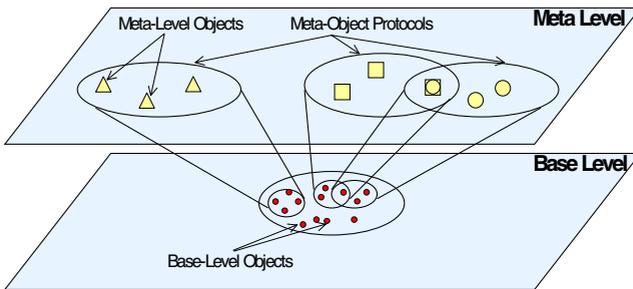

Fig. 1. Base- and meta-level objects.

Although Java supports structural reflection, which provides information about objects and classes at run time, it does not support behavioral reflection, which is needed to dynamically change the interpretation of an application.

TRAP/J and several other projects [6, 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], discussed later, extend Java with behavioral reflection.

2.2   Aspect-Oriented Programming and Java

Many programming aspects, such as quality of service, fault tolerance, and security, cannot be localized in one object or procedure [4]. Typically, the implementation of such an aspect is scattered throughout a program, resulting in *tangled* code that is difficult to develop and maintain. Aspect-oriented programming (AOP) [4] addresses this problem by promoting separation of concerns at development time. Later, at compile or run time, an *aspect weaver* is used to weave different aspects of the program together to form a complete program with the desired behavior. *AspectJ* [17], used in TRAP/J, is a widely used AOP extension to Java. A class-like language element, called an *aspect*, is used to modularize a cross-cutting concern. An aspect typically consists of "pointcuts" and "advice." A *pointcut* defines a set of "join points" in a program, and *advice* is code that executes at each join point defined in a pointcut. As described in Section 4, TRAP/J uses aspects to provide the necessary "hooks" to realize run-time recomposition of the application.

2.3   Related Work

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

The first category includes *middleware extensions* 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. Examples include TAO [18], DynamicTAO [19], ZEN [20], Open ORB [21], QuO [22], Orbix [23], ORBacus [24], Squirrel [25], IRL [26], Eternal [27], and ACT [28]. In addition to providing transparency to the functional code, some approaches provide transparency to the distribution middleware code as well. For example, IRL and ACT use CORBA portable interceptors [29] to intercept CORBA messages transparently, and Eternal 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.

In the second category, *programming language extensions* are introduced to facilitate the development of adaptive code. Extensions to Java include Open Java [11], R-Java [12], Handi-Wrap [16], PCL [30], and Adaptive Java [6]. In general, approaches in this category significantly improve the process for developing and maintaining adaptable programs by hiding the details of interceptions and redirections from the application developer. To add adaptive behavior to an *existing* program, however, a developer is required to modify the program source

code directly. In other words, this approach is well suited to the development of new adaptable applications, but cannot be applied transparently to existing ones.

The third category provides such transparency by *extending virtual machines* with facilities to intercept and redirect interactions in the functional code. Examples of extensions to the Java virtual machine (JVM) include Iguana/J [8], metaXa [9] (previously called Meta Java), Guaraná [13], PROSE [31], and R-Java [12]. These projects employ a variety of techniques. For example, Guaraná extends the JVM by directly modifying the Kaffe open source JVM [32], whereas PROSE and Iguana/J extend the standard JVM by weaving aspects, without modifying the 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 might reduce their portability.

Finally, the fourth category includes approaches that transparently *augment the application code itself* with facilities for interception and redirection. Prominent examples include generative programming, generic programming, feature-oriented development, and AOP [33, 34]. Among these approaches, AOP, particularly when combined with computational reflection, has been applied to a wide variety of systems. Example projects include AspectJ [17], Hyper/J [35], DemeterJ (DJ) [36], JAC [37], Composition Filters [38], ARCAD [7], Reflex [15], Kava [14], Dalang [39], Javassist [40]. 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. TRAP/J is most similar to the RNTL ARCAD project [7]. ARCAD also uses AspectJ at compile time and behavioral reflection at run time. However, the partial behavioral reflection provided in TRAP/J is more fine-grained and efficient 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, described in the next section. Reflex [15] also provides a partial behavioral reflection us-

ing a two-phase approach. However, unlike TRAP/J and ARCAD, Reflex uses *load time* byte code rewriting to weave interception hooks into existing Java programs, resulting in additional overhead. On the other hand, Reflex provides a comprehensive approach to partial behavioral reflection, including selection of classes and/or objects to be reflective, selection of operations to be reified (*e.g., message send*, *message receive*, and *object creation*), and selection of specific operation occurrences to be reified. TRAP/J supports reflection of both classes and objects, but currently supports reification of only *send message* operations. TRAP/J supports *automatic* activation/deactivation of reflection, that is, there is no need for explicit calls to activate/deactivate a specific reification at run time. We argue this feature simplifies the use of TRAP/J relative to other approaches.

3   TRAP/J Operation

The TRAP/J prototype leverages Java structural reflection both in its code generators and in its run-time redirection of messages. For the aspect syntax and the aspect weaver, we adopted *AspectJ* [17].

3.1   Overview

TRAP/J operates in two phases. The first phase takes place at compile time and converts an existing application into an application that is *adapt-ready* [3] with respect to one or more concerns. We say that a program is adapt-ready if its behavior can be managed at run time. Figure 2 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 the 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). We use the term *composer* to refer to the entity that actually uses this adaptation infrastructure to adapt the adapt-ready application. 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. In the current prototype of TRAP/J, the composer is an administrator interacting with the adapt-ready application through graphical user interfaces, called administrative consoles. However, our ongoing investigations address use of TRAP/J by software-based composers.
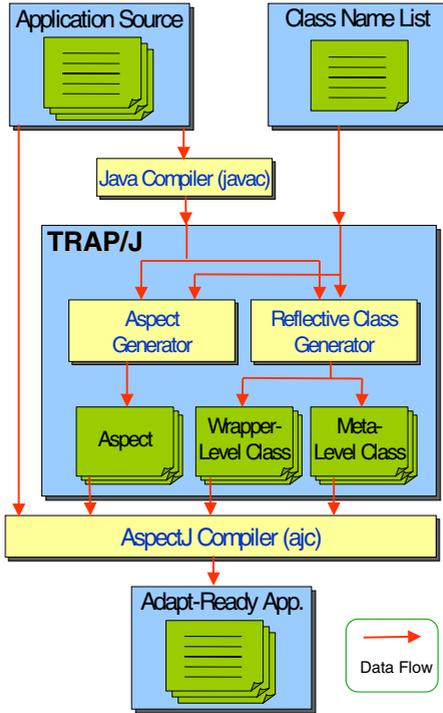
Fig. 2. TRAP/J operation at compile time.

Figure 3 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 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.

3.2 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 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
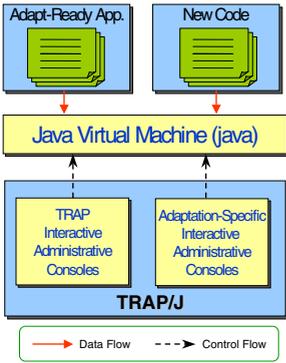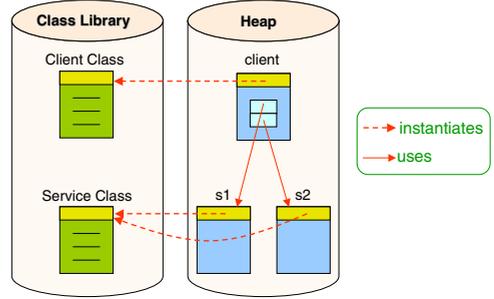
Fig. 3. TRAP/J run-time support.

Fig. 4. A simplified run-time class graph.

are shown using dashed arrows, and the "uses" relationships among objects are depicted with solid arrows.

Figure 5 illustrates a layered run-time class graph model for this application. Note that the base-level layer depicted in Figure 5 is equivalent to the class graph illustrated in Figure 4. For simplicity, only the "uses" relationships are represented in Figure 5. 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. 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 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
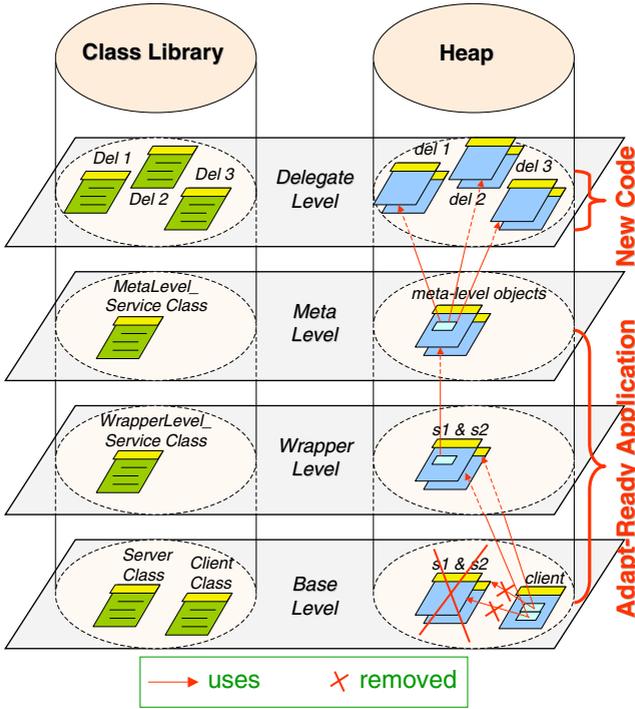
Fig. 5. TRAP layered run-time model.

TRAP/J generators to generate the wrapper and metaobject classes associated with the socket class. Next, at run time, a composer can program the metaobject associated with the socket object to support dynamic reconfiguration. Programming the metaobject can be done by introducing a delegate class to the metaobject at run time. The metaobject then loads the delegate class, instantiates an object of the delegate class, intercepts all subsequent messages originally targeted to the socket object, and forwards the intercepted messages to the delegate object. Let us assume that the delegate object provides a new implementation for the send(...) method of the socket class. In this case, all subsequent messages to the send(...) method are handled by the delegate object and the other messages are handled by the original socket object. Alternatively, the delegate object could modify the intercepted messages and then forward them back to the socket object, resulting in a new behavior. Note that 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 [41] approach to dynamic adaptation.

4   Case Study

To illustrate the use of TRAP/J, we describe a detailed example in the context of a specific application. The example application is a Java program for streaming live audio over a network [42]. Although the original application was developed for wired networks, we used TRAP/J to make it adaptable to wireless environments, where the packet loss rate is dynamic and location dependent. Specifically, we use TRAP/J to weave in an adaptable socket class, whose behavior can be adapted at run time to compensate the packet loss in wireless networks.

4.1   Example Application

The Audio-Streaming Application (ASA) [42] is designed to stream interactive audio from a microphone at one network node to multiple receiving nodes. The program is designed to satisfy a real-time constraint, specifically, the delay between audio recording and playing should be less than 100 milliseconds.

Figure 6 illustrates the operation of ASA in a wireless environment: a laptop workstation transmits the audio stream to multiple wireless iPAQs over an 802.11b (11Mbps) ad-hoc wireless local area network (WLAN). Unlike wired networks, wireless environments 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.
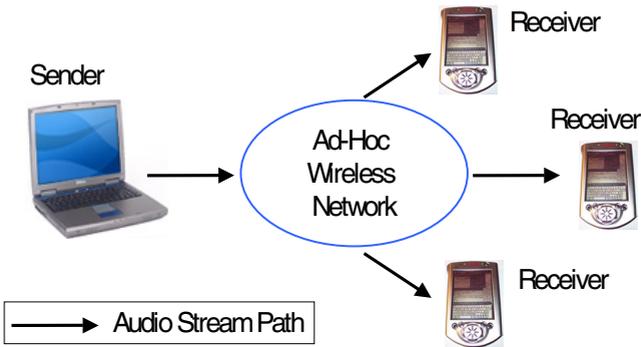

Fig. 6. Audio streaming in a wireless LAN.

Figure 7 illustrates the strategy we used to enable ASA to adapt to variable channel conditions in wireless networks. Specifically, we used TRAP/J to modify ASA transparently so that it uses MetaSockets instead of Java multicast sockets. *MetaSockets* [42] are adaptable communication components created from existing

Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli (*e.g.*, dynamic wireless channel conditions). In an earlier study, we implemented MetaSockets in *Adaptive Java* [6], which extends Java with new constructs and keywords to facilitate the design of adaptable components. In this study, we use TRAP/J to replace normal Java sockets with MetaSockets, transparently to the ASA code.
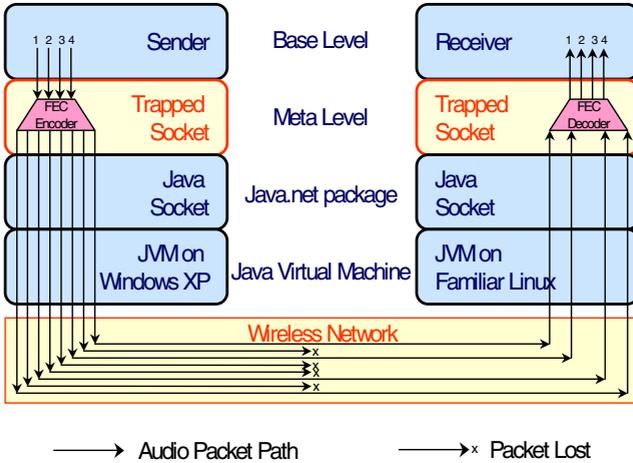

Fig. 7. Adaptation strategy.

The particular MetaSocket adaptation used here is the dynamic insertion and removal of *forward-error correction* (FEC) filters [43]. 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.

4.2   Making ASA Adapt-Ready

Figure 8 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.

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

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

Fig. 8. Excerpted code for the Sender class.

generators. The TRAP/J class generators produce one aspect file, named Absorbing_MulticastSocket.aj (for base level), and two reflective classes, named Wrapper-Level_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 new versions of ajc accept .class files instead of .java files [44], which implies that TRAP/J does 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 9 shows excerpted code for the generated Absorbing_MulticastSocket aspect. This figure shows the "initialization" pointcut (lines 2-3) and its corresponding advice (lines 5-9) for the MulticastSocket constructor used in the Sender class. Referring back to the layered class graph in Figure 5, 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.

Generated Wrapper-Level Class. Figure 10 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 4-5). Also, all the public instance methods are overridden (lines 7-22).

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

Fig. 9. Excerpted generated aspect code.

To better explain how the generated code works, let us consider the details of how the send method is overridden, as shown in Figure 10. The generated send method first checks if the metaObject variable, referring to the metaobject corresponding to this wrapper-level object, is null (line 9). 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 11-21). 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 11 shows excerpted code for Meta-Level_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,

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

Fig. 10. Excerpted generated wrapper code.

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.

4.3   Audio Streaming Experiment

To evaluate the TRAP/J-enhanced audio application, we conducted experiments using the configuration illustrated in Figure 6. Figure 12 shows a sample of the results. An experiment was conducted with an adapt-ready version of ASA. A

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

Fig. 11. Excerpted generated metaobject code.

user holding a receiving iPAQ is walking within the wireless cell, receiving and playing a live audio stream. 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.

5   Summary and Future Investigations

In this paper, we described the design and operation of TRAP/J, a generator framework that enables dynamic reconfiguration of Java applications without modifying the application source code and without extending the JVM. TRAP/J operates in two phases. At compile time, TRAP/J produces an adapt-ready version of the application. Later at run time, TRAP/J enables adding new
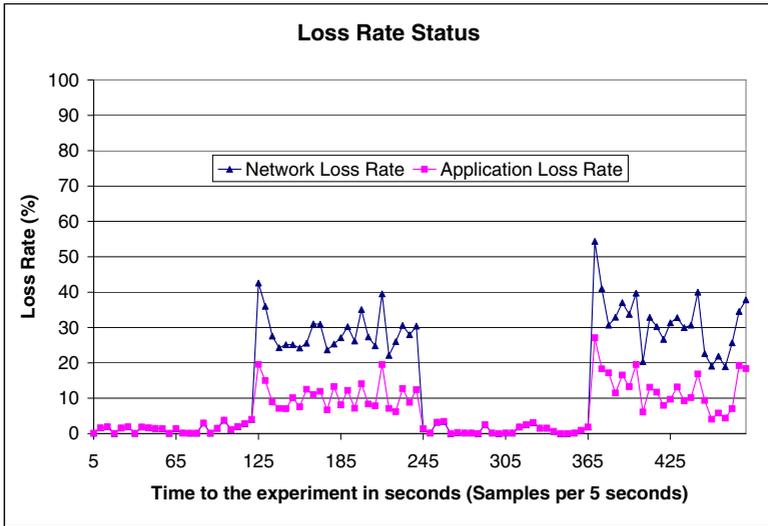
Fig. 12. The effect of using FEC filters to adapt ASA to high loss rates on a wireless network.

behavior to the adapt-ready application dynamically. A case study in a wireless network environment was used to demonstrate the operation and effectiveness of TRAP/J.

Our ongoing investigations involving TRAP address several issues. First, in the current TRAP/J prototype, we addressed the reification of only method invocations at the base level. We are planning to extend the TRAP/J reflective model to include reification of other base-level operations, including object creation, object deletion, method call (send), method dispatch, field read, and field write. Second, the concept used to develop TRAP/J does not depend to the Java Language. Currently, we are developing a TRAP/C++ prototype that enables dynamic reconfiguration of existing C++ programs. For generating adapt-ready programs in TRAP/C++, instead of using an aspect weaver such as AspectJ, we employ a compile-time MOP using Open C++ [45]. Third, TRAP can be used to support autonomic computing [46], where the behavior of *manageable elements* of a program can be externally controlled by software-based composers. Specifically, TRAP can be used to wrap existing applications transparently to generate such manageable elements. Fourth, the TRAP approach can be used for transparent and adaptive auditing of software. The foundation of an effective covert auditing system is the ability to modify the behavior of software components at run time, namely, to insert and remove software sensors (and possibly actuators) in active components, while prohibiting arbitrary unauthorized (and possibly malicious) code from loading and executing as insider. Currently, we are investigating this application of TRAP for critical infrastructure protection.

Further Information

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

References

1. Maes, P.: Concepts and experiments in computational reflection. In: Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA), ACM Press (1987) 147–155
2. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of Metaobject Protocols. MIT Press (1991)
3. Yang, Z., Cheng, B.H., Stirewalt, R.E.K., Sowell, J., Sadjadi, S.M., McKinley, P.K.: An aspect-oriented approach to dynamic adaptation. In: Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02). (2002)
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241 (1997)
5. Sadjadi, S.M., McKinley, P.K., Stirewalt, R.E.K., Cheng, B.H.: Generation of self-optimizing wireless network applications. In: Proceedings of the International Conference on Autonomic Computing (ICAC-04), New York, NY (2004) 310–311
6. Kasten, E.P., McKinley, P.K., Sadjadi, S.M., Stirewalt, R.E.K.: Separating introspection and intercession in metamorphic distributed systems. In: Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02), Vienna, Austria (2002) 465–472
7. David, P.C., Ledoux, T., Bouraqadi-Saadani, N.M.N.: Two-step weaving with reflection using AspectJ. In: OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa (2001)
8. Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behaviour. In: Proceedings of the 16th European Conference on Object-Oriented Programming. (2002)
9. Golm, M., Kleinoder, J.: metaXa and the future of reflection. In: Proceedings of Workshop on Reflective Programming in C++ and Java. (1998) 1–5
10. Wu, Z.: Reflective Java and a reflective component-based transaction architecture. In: Proceedings of Workshop on Reflective Programming in C++ and Java. (1998)
11. Tatsubori, M., Chiba, S., Itano, K., Killijian, M.O.: OpenJava: A class-based macro system for Java. In: Proceedings of OORaSE. (1999) 117–133

12. de Oliveira Guimarães, J.: Reflection for statically typed languages. In: Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98). (1998) 440–461
13. Oliva, A., Buzato, L.E.: The implementation of Guaraná on Java. Technical Report IC-98-32, Universidade Estadual de Campinas (1998)
14. Welch, I., Stroud, R.J.: Kava - A Reflective Java Based on Bytecode Rewriting. In Cazzola, W., Stroud, R.J., Tisato, F., eds.: Reflection and Software Engineering. Lecture Notes in Computer Science 1826. Springer-Verlag, Heidelberg, Germany (2000) 157–169
15. Tanter, É., Noyè, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R., Steele, Jr., G.L., eds.: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003), Anaheim, California, ACM Press (2003) 27–46
16. Baker, J., Hsieh, W.: Runtime aspect weaving through metaprogramming. In: Proceedings of the first International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands (2002)
17. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science **2072** (2001) 327–355
18. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Computer Communications **21** (1998) 294–324
19. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: 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 (2000)
20. Klefstad, R., Schmidt, D.C., O'Ryan, C.: Towards highly configurable real-time object request brokers. In: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (2002)
21. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: 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 (1998)
22. Zinky, J.A., Bakken, D.E., Schantz, R.E.: Architectural support for quality of service for CORBA objects. Theory and Practice of Object Systems **3** (1997)
23. IONA Technology: (Orbix) Available at URL: `http://www.iona.com/products/orbix.htm`.
24. IONA Technologies Inc.: ORBacus for C++ and Java version 4.1.0. (2001)
25. Koster, R., Black, A.P., Huang, J., Walpole, J., Pu, C.: Thread transparency in information flow middleware. In: Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer Verlag (2001)
26. Baldoni, R., Marchetti, C., Termini, A.: Active software replication through a three-tier approach. In: Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02), Osaka, Japan (2002) 109–118
27. Moser, L., Melliar-Smith, P., Narasimhan, P., Tewksbury, L., Kalogeraki, V.: The Eternal system: An architecture for enterprise applications. In: Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99). (1999)
28. Sadjadi, S.M., McKinley, P.K.: ACT: An adaptive CORBA template to support unanticipated adaptation. In: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04), Tokyo, Japan (2004)

29. Object Management Group Framingham, Massachusett: The Common Object Request Broker: Architecture and Specification Version 3.0. (2003) Available at URL: `http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf`.
30. Adve, V., Lam, V.V., Ensink, B.: 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 (2001)
31. Popovici, A., Gross, T., Alonso, G.: Dynamic homogenous AOP with PROSE. Technical report, Department of Computer Science, Federal Institute of Technology, Zurich (2001)
32. Mumtaz, S., Ahmad, N.: (Architecture of kaffe) Available at URL: `http://wiki.cs.uiuc.edu/cs427/Kaffe+Architecture+Project+Site`.
33. Czarnecki, K., Eisenecker, U.: Generative programming. Addison Wesley (2000)
34. Tarr, P., Ossher, H., eds.: Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (W17). (2001)
35. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. Communications of the ACM **44** (2001) 43–50
36. Lieberherr, K., Orleans, D., Ovlinger, J.: Aspect-oriented programming with adaptive methods. Communications of the ACM **44** (2001) 39–41
37. Pawlak, R., Seinturier, L., Duchien, L., Florin., G.: JAC: A flexible and efficient solution for aspect-oriented programming in Java. In: Proceedings of Reflection 2001, LNCS 2192. (2001) 1–24
38. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Communications of ACM (2001) 51–57
39. Welch, I., Stroud, R.: Dalang — a reflective extension for java. Technical Report CS-TR-672, University of Newcastle upon Tyne, East Lansing, Michigan (1999)
40. Chiba, S.: Load-time structural reflection in Java. Lecture Notes in Computer Science **1850** (2000)
41. Piveta, E.K., Zancanella, L.C.: Aspect weaving strategies. Journal of Universal Computer Science **9** (2003) 970—983
42. Sadjadi, S.M., McKinley, P.K., Kasten, E.P.: 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), San Juan, Puerto Rico (2003) 46–55
43. Rizzo, L., Vicisano, L.: RMDP: An FEC-based reliable multicast protocol for wireless environments. ACM Mobile Computer and Communication Review **2** (1998)
44. Colyer, A.: (Aspectj 1.2 released) Available at URL: `http://www.theserverside.com/news/thread.tss?thread_id=26144`.
45. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture. Lecture Notes in Computer Science **707** (1993)
46. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36** (2003) 41–50