TRAP: Transparent Reflective Aspect Programming

S. M. Sadjadi, P. K. McKinley, R. .E. K. Stirewalt, and B. H. C. Cheng

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824

{sadjadis,mckinley,stire,chengb}@cse.msu.edu

ABSTRACT

This paper introduces transparent reflective aspect programming (TRAP), a generator framework to support efficient, dynamic, and traceable adaptation in software. TRAP enables adaptive functionality to be added to an existing application without modifying its source code. To reduce overhead, TRAP enables the developer to select, at compile time, a subset of classes to support adaptation through runtime aspect weaving. TRAP uses aspect-oriented programming and behavioral reflection to automatically generate the required aspects and reflective classes associated with the selected types. At run time, new adaptive behavior can be introduced to the application transparently with respect to the original code. TRAP can be applied to any object-oriented language that supports structural reflection. A prototype, TRAP/J, which has been developed for use with Java applications, is described. A case study is presented in which TRAP was used to enable an existing audio-streaming application to operate effectively in a wireless network environment by adapting to changing conditions.

Keywords

Dynamic adaptation, aspect-oriented programming, behavioral reflection, adaptive middleware, transparent adaptation, quality-of-service, mobile computing.

1. INTRODUCTION

As the computing and communication infrastructure continues to expand and diversify, the need for dynamic adaptation in software is increasing. For example, the advent of the "Mobile Internet" has produced a variety of handheld and wearable computing devices, whose software must adapt to multiple, potentially conflicting concerns, such as quality-of-service, security, and energy consumption. Unfortunately, many applications being ported to mobile computing environments were not designed to adapt to the corresponding concerns. We say that a program is *adaptable* if it contains facilities for selecting and incorporating new behaviors at run time. Modifying programs directly to support adaptation can lead to tangled code that is error-prone and difficult to maintain. This paper introduces transparent reflective aspect programming (TRAP), which combines behavioral reflection [1,2] and aspect-oriented programming [3] to transform extant programs into new programs that are adaptable to specific concerns. The proposed approach is novel in that it introduces only the minimum infrastructure required to support the desired adaptations, while requiring no modification to the application source code.

Adaptation in software requires some mechanism to support changes in behavior. The predominant mechanism for implementing adaptation in object-oriented software is *behavioral reflection*, which can be used to modify how an object responds to a message. In recent years, behavioral reflection has been used to support adapation to a variety of different concerns, including quality of service [4–10] and fault tolerance [11–13]. Unfortunately, programs that use behavioral reflection incur additional overhead, as in some cases every message sent to an object must be intercepted and possibly redirected. To provide efficient adaptation, a developer should be able to *selectively* introduce behavioral reflection only where needed to support the desired adaptations. Specifically, the developer should be able to identify, at compile time, which individual classes should be able to support dynamic adaptation at run time.

In an earlier paper [14], 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 support classes and supporting infrastructure for adaptation, and must create an aspect that weaves this infrastructure into the existing program. TRAP extends and generalizes this approach in two dimensions. First, TRAP supports general behavioral reflection by automatically generating wrapper classes and meta-classes from selected classes in an application. Intuitively, an instance of a wrapper class intercepts messages that were destined for an instance of the wrapped class and can redirect these messages to a meta object, whose behavior is programmable at run time. Second, TRAP generates aspects that replace instantiations of selected classes with instantiations of their corresponding wrapper classes. This two-pronged, automated approach enables powerful behavioral reflection with minimal overhead.

To validate these ideas, we developed TRAP/J, which instantiates TRAP for Java programs. TRAP/J provides two generators, both of which take as input Java class files and a list of selected classes. For each selected class, one generator produces source files that define the corresponding wrapper- and meta-classes, while the other produces an aspect in the AspectJ language [15]. Next, the generated aspects and reflective classes, along with the original application source code, are woven together. The resulting *adapt-ready* application can be augmented at run time with new, adaptive code that modifies the behavior of the selected classes on a per-object basis. TRAP/J adds minimal overhead (one level of call forwarding for only those selected classes) while supporting dynamic adaptation in legacy Java applications.

The remainder of this paper is organized as follows. Section 2 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 provides concluding remarks and discusses possible future research directions.

2. BACKGROUND

Many research groups have addressed the development of adaptive software. We categorize these efforts into two major types: those that are not transparent to the application source code, and those that are transparent.

2.1 Non-Transparent Adaptation

If the required adaptation is anticipated during application development, then the adaptation code can be developed and integrated directly with the application code. Three general approaches have been used to realize adaptation that is visible to the application code. First, the adaptive behavior can simply be "hard-wired" into the application, using *conventional* programming languages. For example, recently a wide variety of *context-aware* systems and toolkits [16] have been developed, where software execution flow is directly affected by the external environment. Second, the programming language might be extended to provide more convenient (and in some cases more powerful) constructs and keywords for adaptation. Approaches based on language extensions include 3-LISP [2], 3KRS [1], CLOS [17], OpenC++ [18], ABCL-R2 [19], AL-1/D [20], PCL [21], Hadas [22], and Adaptive Java [23]. Third, the new libraries and APIs can augment a language to provide better monitoring and control of the platform on which the application is running. Examples of this approach include adaptive middleware frameworks such as QuO [24] and Open ORB [5], as well as adaptive network interface components, such as Rocks [25] and MetaSockets [10].

Unfortunately, when adaptive behavior is not transparent, the resulting adaptive code is likely to be tangled with the application code. Moreover, the adaptations not anticipated at development time might be difficult to add later. Indeed, the application source code may not even be available for later modifications.

2.2 Transparent Adaptation

To provide adaptation transparently with respect to the application code, many approaches use "interception" and "redirection" to support weaving of adaptive code with the original application. Proposed solutions differ in where the interception takes place.

One method is to extend or enhance the run-time environment, including run-time libraries, run-time interpreters, and operating systems. Examples of extending a run-time interpreter include Iguana/J [26], Meta Java [27], Guaraná on Java [28], and PROSE [29]. While providing transparency, this approach reduces portability and can produce significant performance overhead. For example, according to [26], the time for common operations such as creating new objects can be increased by an order of magnitude. Finally, this approach is only possible when the run-time environment is accessible for modification.

A second method is to introduce adapability in a supporting middleware platform. Middleware can support adaptability in various concerns, including fault-tolerance, quality-of-service, security, and energy consumption. Examples include TAO [30], DynamicTAO [8], ZEN [31], Squirrel [32], IRL [11], Eternal [12], and ACT [33]. However, as with extensions to a run-time environment, middleware-based approaches can introduce significant processing overhead. For example, ACT [33] introduces overhead for unmarshalling and marshalling the intercepted requests inside the ORB, whereas ZEN [31] introduces delay for loading of middleware components by way of the virtual component pattern [34]. Moreover, using this method to provide adaptability to a legacy application requires that the application was developed originally with an appropriate middleware platform, or that it be ported to one.

The third method to transparent adaptation, and the one used in TRAP, is to augment the application itself with infrastructure to support adaptation. In the past few years, numerous techniques have been proposed to enable advanced separation of concerns (ASOC) [35]. Prominent examples include domain-specific languages, generative programming, generic programming, constraint languages, feature-oriented development, and aspect-oriented programming [36].

Among all ASOC approaches, aspect-oriented programming, especially when combined with computational reflection [1, 2], has received perhaps the most attention. AspectJ [15], Hyper/J [37], DemeterJ (DJ) [38], JAC [39], Kava [40], and Composition Filters [41] can weave adaptive code into an application. Most of these systems are designed to introduce adaptive behavior at compile time. To enable the application to be extended at run time, a two-phase approach can be used [14, 42]. In the first phase, interception hooks are prepared at compile time using static weaving of aspects. In the second phase, intercepted operations are forwarded to adaptive code using reflection. Like [14, 42], TRAP employs selective adaptation in two phases, which makes it both transparent and efficient. However, TRAP also provides a *systematic* process in which the adaptation infrastructure and aspects for weaving the infrastructure into the application code are generated automatically. Moreover, the adaptation is *traceable* in the sense that the application can

always return to its original behavior, if required. In the remainder of this paper, we describe the TRAP/J prototype and the details of the techniques used to provide selective trade-offs between adaptation flexibility and efficiency.

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* [15].

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* [14] with respect to one or more concerns. Figure 1 shows a high-level representation of the operation of TRAP/J at compile time. The application source code is compiled using the Java language 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 to produce the adapt-ready application.

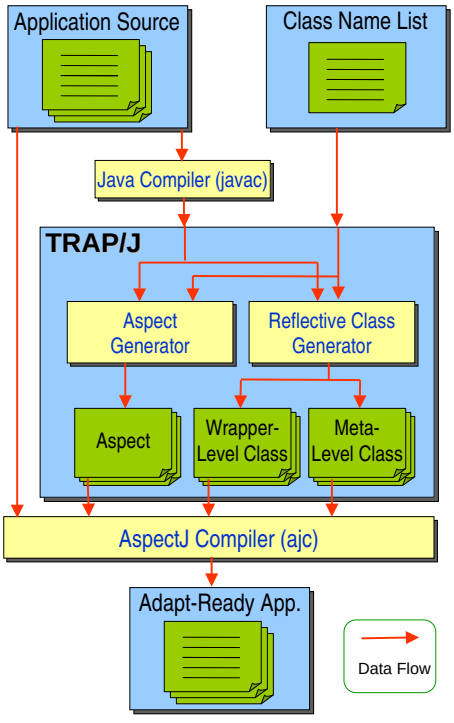

Figure 1: TRAP/J operation at compile time.

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), as well as a set of interactive administrative consoles. Figure 2 illustrates the interaction among the Java Virtual Machine (JVM) and the interactive administrative consoles. First, the adapt-ready application is loaded by JVM. Next, if an adaptation is required, an interactive administrative console enables a user to dynamically add new code to the adapt-ready application. As part of the behavioral reflection provided in the adaptation infrastructure, a meta-object protocol (MOP) is supported by TRAP/J that allows the loading of new code for changing the behavior of the adapt-ready application. Depending on the specific adaptation, an application-specific console may be required to control the adaptation.

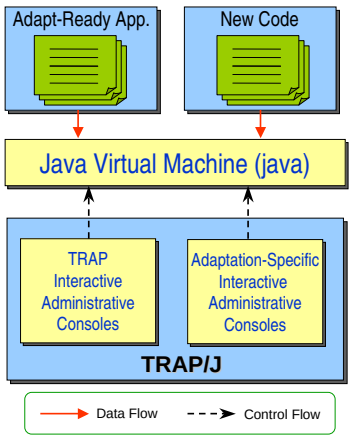

Figure 2: TRAP/J run-time support.

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 3 depicts a simple run-time class graph for this application that is compliant with the run-time architecture of most 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 shown using solid arrows.

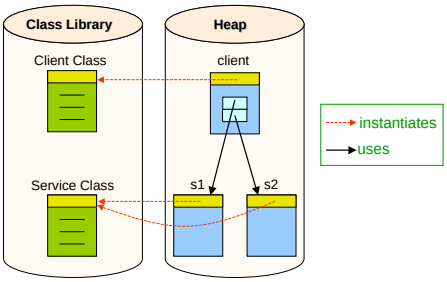

Figure 3: A simplified run-time class graph.

Figure 4 illustrates a layered run-time class graph model for this application. Note that the base-level layer depicted in Figure 4 is equivalent to the class graph illustrated in Figure 3. For simplicity, only the "uses" relationships are represented in Figure 4. The wrapper level contains the generated wrapper classes for the selected subset of base-

level classes and their corresponding instances. These instances are used by the base-level client objects 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. Associated with each wrapper object can be at most one meta-object. Note that each meta-object is dynamically programmable. In other words, the behavioral reflection provided by TRAP/J has "object-level" granularity, as opposed to class- or message-level granularity provided by other reflective paradigms.

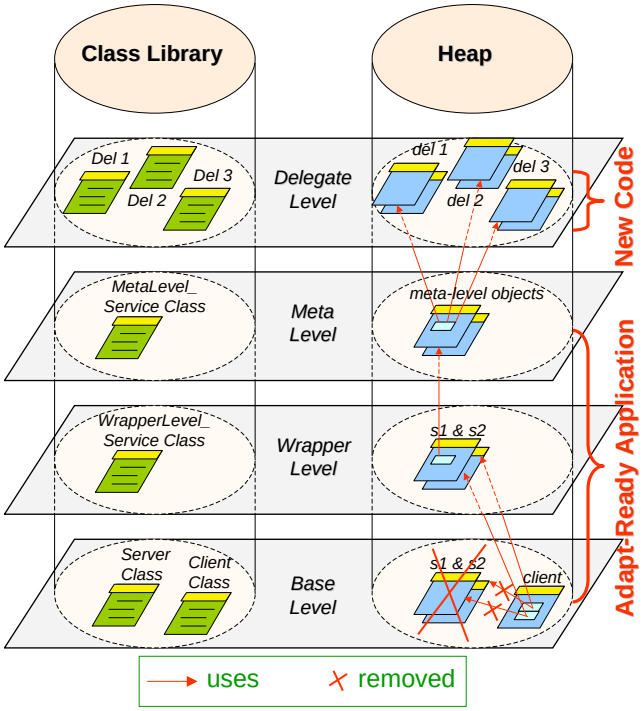

Figure 4: TRAP layered run-time model.

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 to 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. Using TRAP/J interactive administrative consoles, meta-objects can be programmed dynamically to redirect messages destined originally to base-level methods to their corresponding implementations in delegate classes. Each meta-object 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 meta-objects.

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 [10]. 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.

4.1 Example Application

The Audio-Streaming Application (ASA) [10] 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, that the delay between audio recording and playing it should be less than 100 milliseconds.

Wired Environment. Figure 5 illustrates operation of ASA in a wired network; a desktop workstation transmits a live audio stream to multiple desktop workstations over a 100Mbps wired local area network (LAN). ASA has two parts. The *sender* records the audio from a microphone at a rate of 8000 one-byte samples per second. Every 500 samples (62.5 milliseconds of recorded audio) are sent as one packet to the receivers using IP multicast. The reason for choosing a packet length of 500 bytes, and not a smaller size, is that the sender uses the Java sound package, which delivers recorded audio every 500 bytes. The receivers play audio packets as soon as they arrive.

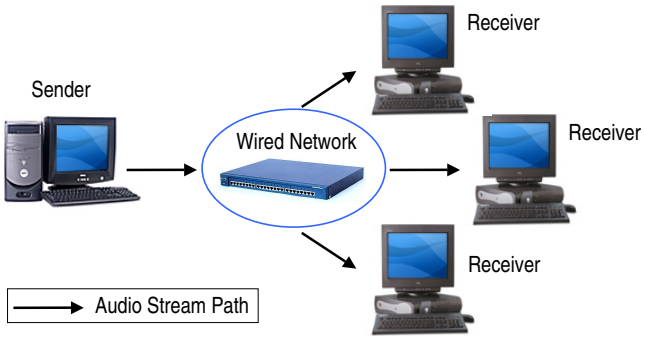

Figure 5: Audio streaming in a wired LAN.

Wireless Environment. Figure 6 illustrates the operation of ASA in a wireless environment: a desktop workstation transmits the audio stream to multiple wireless laptops over an 802.11b (11Mbps) wireless local area network (WLAN). Unlike wired networks, in 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.

Adaptation Strategy. Figure 7 illustrates the strategy we used to enable ASA to adapt to variable channel conditions in wireless networks. We used TRAP/J to modify ASA transparently so that it uses *MetaSockets* instead of Java multicast sockets. A MetaSocket is a low-level middleware mechanism developed in our group in an earlier study [10]. MetaSockets can be configured at run-time with *filters*, which intercept and massage the data stream traversing the socket. In our earlier study, we implemented MetaSock-

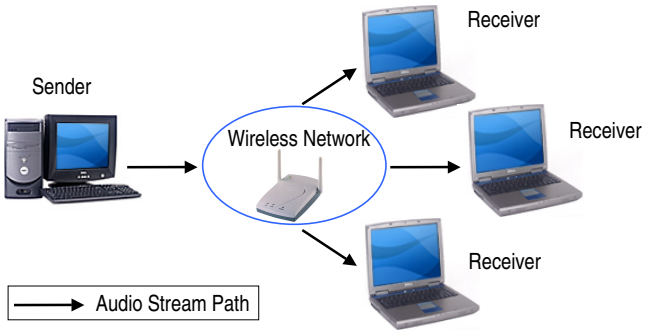

Figure 6: Audio streaming in a wireless LAN.

ets in Adaptive Java [23], an extension to Java that supports dynamic run-time adaptation, but is obviously not transparent. In this study, we use TRAP/J to replace normal Java sockets with MetaSockets, transparently to the ASA code.

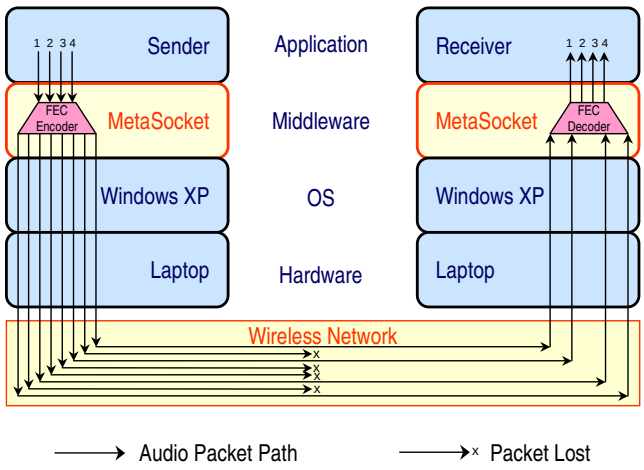

Figure 7: Adaptation strategy.

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

The FEC codes used in this study are (n, k) block erasure codes, which convert k packets into n encoded packets such that any k encoded packets can be used to reconstruct the k source packets [43]. These codes are lossless in that a successful decoding produces exactly the original data. We use only systematic codes, which means that the first k of the n encoded packets are identical to the original k packets. We refer to the first k packets as data packets, and the remaining $(n-k)$ packets as parity packets. Each set of n encoded packets is referred to as a group. Figure 7 shows an example using an $(8, 4)$ code. As long as at least 4 of the 8 packets arrive intact, the receiver can reconstruct the original 4 data packets.

The advantage of using block erasure codes for multicasting is that a single parity packet can be used to correct independent single-packet losses among different receivers [44]. Recently, Rizzo [44] studied the feasibility of software encoding and decoding for packet-level FEC, using a particular block erasure code called the Vandermonde code. We used an open-source Java implementation of Rizzo's FEC library, available from Swarmcast [45]. In general, we found the Swarmcast library to provide a convenient interface and good performance. Next, we describe how we used TRAP/J to make the original ASA application into one that supports dynamic adaptation.

4.2 Making ASA Adapt-Ready

Figure 8 shows excerpted code for the 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, on each iteration, reads 500 bytes of live audio and transmits them 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 ...
 }

 public static void main(String[] args)
 { ...
  Sender sender = new Sender();
  sender.run(); ...
 } ...
} // end Sender
```

Figure 8: Excerpted code for the Sender class.

Compile-Time Actions. Figure 9 sender part of ASA. 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, and two reflective classes, named WrapperLevel_MulticastSocket.java and MetaLevel_MulticastSocket.java. Next, the generated files and the original application source code are compiled using the AspectJ compiler (ajc). 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.

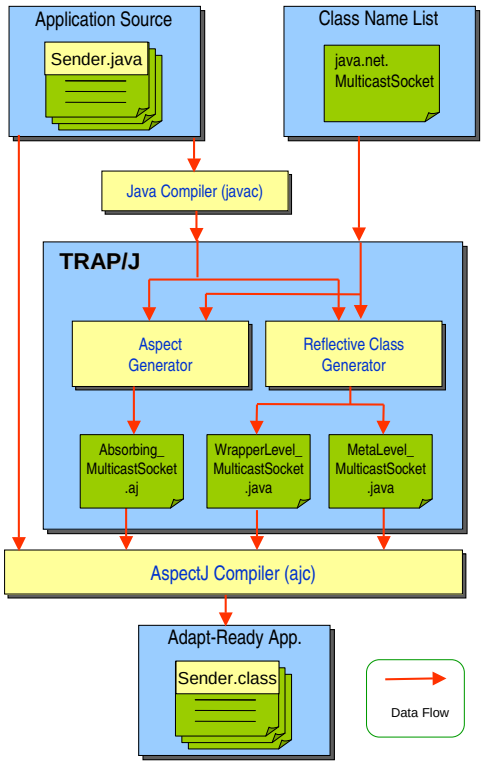

Figure 9: Compiling steps for Sender program in TRAP/J.

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 10 shows excerpted code for the generated Absorbing_MulticastSocket aspect. This figure shows the "initialization" pointcut and its corresponding advice for the MulticastSocket constructor used in the Sender class. Referring back to the layered class graph in Figure 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.

Generated Wrapper-Level Class. Figure 11 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). Also, all the public instance methods are overridden.

To better explain how the generated code works, let us walk through the details of how the send method is overridden, as shown in Figure . The generated send method first checks if the metaObject variable, referring to the meta-level object corresponding to this wrapper-level object, is null. 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. If the metaObject variable

```
public aspect Absorbing_MulticastSocket
{
  pointcut MulticastSocket() :
    call(java.net.MulticastSocket.new()) && ...;

  java.net.MulticastSocket around()
    throws java.net.SocketException
    : MulticastSocket()
  {
    return new WrapperLevel_MulticastSocket();
  }

  pointcut MulticastSocket_int(int p0) :
    call(java.net.MulticastSocket.new(int)) && args(p0) &&
    ...;

  // Pointcuts and advices around the final public methods
   pointcut  getClass(WrapperLevel_MulticastSocket  targe-
tObj) :
    ...;
}
```

Figure 10: Excerpted code for Absorbing_MulticastSocket, the generated aspect for MulticastSocket.

is not null, then Java reflection is used to create an object of type Method that includes necessary information on the method being called at the wrapper-level, to be passed to the meta-level object (metaObject) through the invokeMetaMethod method. In addition to this method information, the arguments passed to the wrapper-level method and an isReplyReady boolean object are also passed to the invokeMetaMethod. The reason for passing the extra variable, isReplyReady, is that the invokeMetaMethod can set isReplyReady to TRUE in order to inform the wrapper-level method that the reply to the request is provided by the meta-level object. In this case, the returned results are returned by the wrapper-level method without involving the corresponding base-level method. If this variable is set to FALSE, then it means that the wrapper method still needs to call the corresponding base-level method to acquire the reply. In this case, the arguments may or may not have been changed by the meta-level method.

Next, the wrapper-level method calls the corresponding base-level method. It is possible that the invokeMetaMethod method throws an exception. If the exception type is defined by the base-level method, then the exception is simply thrown by the wrapper-level method. On the other hand, the exception might be of type MetaMethodIsNotAvailable, which means that the meta-level object is not programmed to reply to this message. In this case, the exception is handled gracefully by the wrapper-level method and because isReplyReady is set to FALSE by default, the corresponding base-level method is called. It might be the case that a meta-level object is programmed to reply a message and, as part of its operation (discussed next), 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 a "Orig_" prefix.

```
public class WrapperLevel_MulticastSocket extends
  MulticastSocket implements WrapperLevel_Interface
{
  // Overriding the base-level constructors.
  public WrapperLevel_MulticastSocket()
    throws SocketException { super(); }

  // Overriding the base-level methods.
  public void send(java.net.DatagramPacket p0)
    throws IOException
  {
    if(metaObject == null)
    { super.send(p0); return; }
    ...
    Class[] paramType = new Class[1];
    paramType[0] = java.net.DatagramPacket.class;
    Method method = WrapperLevel_MulticastSocket.class.
      getDeclaredMethod("send", paramType);

    Object[] tempArgs = new Object[1];
    tempArgs[0] = p0;
    ChangeableBoolean isReplyReady =
      new ChangeableBoolean(false);

    try
    {
      metaObject.invokeMetaMethod
        (method, tempArgs, isReplyReady);
    }
    catch (java.io.IOException e) { throw e; }
    catch (MetaMethodIsNotAvailable e) {}

    if(!isReplyReady.booleanValue())
      super.send(p0);
  }

  // Providing access to base-level methods using "Orig_*"
  public void Orig_send(java.net.DatagramPacket p0)
    throws IOException
  { super.send(p0); return; }
}
```

Figure 11: Excerpted code for WrapperLevel_MulticastSocket, the generated wrapper-level class for MulticastSocket.

Generated Meta-Level Class. Figure 12 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 meta-level object 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 is a class that has corresponding wrapper- and meta-level classes, generated by TRAP/J at compile time. Meta-level objects can be programmed dynamically by inserting or removing delegate objects at run time. To enable a user to change the behavior of a meta-level object dynamically, the meta-level class implements the DelegateManagement interface, which in turn extends the Java RMI Remote interface. Using an interactive management console, a user can remotely "program" a meta-level object through Java RMI remote method invocations. The insertDelegate and removeDelegate methods are developed for this purpose.

```
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
  { ...delegates.add(i, delegate); ...}

  public synchronized void removeDelegate(int i)
    throws RemoteException
  { delegates.remove(i); }
  ...

  public synchronized Object invokeMetaMethod
    (Method method, Object[] args,
    ChangeableBoolean isReplyReady) throws Throwable
  {
    if (delegates.size() == 0)
      throw new MetaMethodIsNotAvailable();

    Class[] paramType=method.getParameterTypes();
    Class[] newParamType=new Class[paramType.length+1];
    for (int i = 0; i ¡ paramType.length; i++)
      newParamType[i] = paramType[i];
    newParamType[paramType.length] =
      ChangeableBoolean.class;
    Object[] tempArgs = new Object[args.length+1];
    for(int i=0; i¡args.length; i++)
      tempArgs[i] = args[i];
    tempArgs[args.length] = isReplyReady;
    ...

    // 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 12: Excerpted code for MetaLevel_MulticastSocket, the generated meta-level class for MulticastSocket.

The meta-object protocol developed for meta-level classes defines only one method, invokeMetaMethod, which first checks if any delegate is associated with this meta-level object. 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 Dynamic Adaptation

TRAP/J enables a user to dynamically interact with an application by changing the behavior of meta-level objects at

run time. To do so, the user assigns new delegate classes to the meta-level objects. For this purpose, we developed a simple GUI, Delegate Management Console, illustrated in Figure 13. The user first acquires a reference to a meta-level object from an rmiregistry using the name that the meta-level object used to register itself. Next, the user can insert or remove a delegate in the meta-level object by providing the fully-qualified name of the delegate class. As Figure13 suggests, we can insert a delegate of type Delegate_MulticastSocket_send into the meta-level object referred to by ms in the Sender program and registered with the rmiregistry as Sender_MetaLevel_MulticastSocket_0.

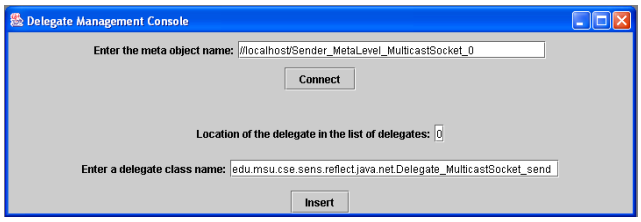

Figure 13: Delegate Management Console.

Figure 14 shows excerpted code for the Delegate_MulticastSocket_send class. To override a base-level method, a delegate needs to implement the same method. A delegate does not need to know the type of the base-level class, unless a delegate method needs to call the original base-level method directly. Moreover, a delegate does not need to implement all the methods of the base-level class, but rather only for those methods to be overridden. For example, in the delegate example shown in Figure 14, only the send method of the MulticastSocket is overridden. This flexibility enables development of delegates for different cross-cutting concerns. Each of the delegates would override some subset of base-level methods, and these subsets may overlap. These separately developed concerns can work together as different delegates that can be assigned to the same meta-level object.

Considering the adaptation required for ASA, we provide a *MetaSocket* to override the send method of ms in the Sender program. The MetaSocket implements a pipeline of filters and enables new filters to be inserted or removed dynamically in response to changing environmental conditions. Figure 14 shows the delegate developed for the send method (the delegate for the receive method is the counterpart of this delegate). The delegate-level send method first obtains the data from dp, the datagram packet sent to the delegate method. Next, it checks to see if there is any filter associated with this MetaSocket. After all filters have processed a packet, the delegate method uses the original base-level send method (implemented as part of the java.net.MulticastSocket, and which can be accessed using the base-level Orig_send method) to transmit the processed datagram packet. Note that because the developer of the delegate knows how to access the original base-level method through corresponding base-level "Orig_*" methods, this invocation does not need to be dynamic using Java reflection. Hence, the performance overhead is small.

Figure 15 shows the Filter Management Console, a MetaSocket-specific management console that enables a user to dynami-

```
public class Delegate_MulticastSocket_send
  extends UnicastRemoteObject
  implements Delegate_Interface, FilterManagement
{
  private Vector filters = null;
  ...
  public synchronized void send(DatagramPacket dp,
    ChangeableBoolean isReplyReady) throws IOException
  {
    Packet packet = new Packet(
      0, new byte[0], dp.getData());
    Packet[] packetList = new Packet[1];
    packetList[0] = packet;
    int filterCounter = 0;
    while (filterCounter ¡ filters.size())
    {
      Filter filter = (Filter)filters.get(filterCounter);
      try
      {
        packetList = filter.process(packetList);
      }
      catch (FilterMismatchException e)
      {
        EventMediator.instance().notify(
          new FilterMismatchEvent(this, e));
        filterCounter–;
      }
      filterCounter++;
    }
    ...
    dp.setData(packet.getByteArray());
    baseObject.Orig_send(dp);
    isReplyReady.value = true;
  }
}
```

Figure 14: Excerpted code for Delegate_MulticastSocket_send.

cally insert or remove filters on a MetaSocket. As this figure suggests, we can dynamically insert a forward-error correction encoder (FECEncoder) to ms of the Sender program at run time to compensate the lower quality-of-service due to noisy conditions on a wireless channel. This insertion could also be automated, as we have done for the Adaptive Java implementation of MetaSockets [10].

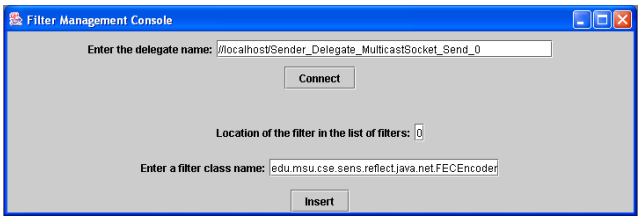

Figure 15: Filter Management Console.

4.4 Sample Experiment

To evaluate the TRAP/J-enhanced audio application, we conducted experiments using the configuration illustrated in Figure 6. Figure 16 shows a sample of the results. An experiment was conducted with an adapt-ready version of ASA. A user holding a receiving laptop 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 inserts the a (20,4) 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. This experiment demonstrates the utility of TRAP/J to transparently enhance an application with new adaptive behavior.

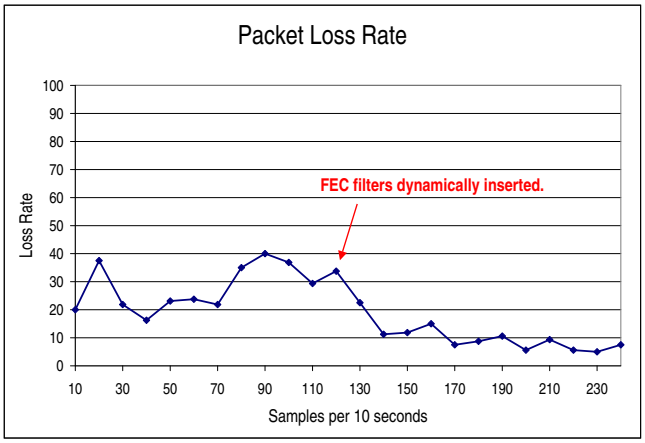

Figure 16: The effect of using FEC filters to adapt ASA to wireless network high loss rate.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced TRAP/J, a generator framework that enables dynamic adaptation in Java applications without modifying the source code. TRAP/J operates in two phases. During compile time, TRAP/J produces an adapt-ready version of the application. Later at run time, TRAP/J enables a user to dynamically add new behavior to the application. A case study in a wireless network environment was used to demonstrate the operation and effectiveness of TRAP/J. The concept used to develop TRAP/J does not depend to the Java Language. In fact, TRAP can be applied to any object-oriented language that provides structural reflection. Our ongoing investigations address (1) the use of TRAP/J to support assurance in adaptation and (2) assisting the developer in identifying affected classes with respect to specified concerns.

6. FURTHER INFORMATION.

A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`.

7. ACKNOWLEDGEMENTS

We are also thankful to Laura Dillon, Farshad Samimi, Eric Kasten, Zhenxiao Yang, Zhinan Zhou, Ji Zhang, and Jesse Sowell for their feedback and their insightful discussions on this work. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation grants CCR-9912407, EIA-0000433, EIA-0130724, and ITR-0313142.

8. REFERENCES

[1] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, December 1987.

[2] B. C. Smith, "Reflection and semantics in Lisp," in *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

[3] 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.

[4] G. Blair, G. Coulson, and N. Davies, "Adaptive middleware for mobile multimedia applications," in *Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 259–273, 1997.

[5] 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.

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

[7] T. Ledoux, "OpenCorba: A reflective open broker," *Lecture Notes in Computer Science*, vol. 1616, 1999.

[8] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, "Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, (New York), April 2000.

[9] M. Roman, F. Kon, and R. H. Campbell, "Reflective middleware: From your desk to your hand," *IEEE Distributed Systems Online*, vol. 2, no. 5, 2001.

[10] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten, "Architecture and operation of an adaptable communication substrate," in *Proceedings of the Ninth International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, May 2003.

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

[12] 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.

[13] J. C. Fabre and T. Perennou, "A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.

[14] Z. Yang, B. Cheng, R. 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*, Nov. 2002.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[16] G. Chen and D. Kotz, "A survey of context-aware mobile computing research," Tech. Rep. TR2000-381, Computer Science Department, Dartmouth College, Hanover, New Hampshire, November 2000.

[17] G. Kiczales, J. d. Rivieres, and D. G. Bobrow, *The Art of Metaobject Protocols*. MIT Press, 1991.

[18] S. Chiba, "A study on a compile-time metaobject protocol," 1996.

[19] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa, "Object-oriented concurrent reflective languages can be implemented efficiently," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (A. Paepcke, ed.), vol. 27, (New York, NY), pp. 127–144, ACM Press, 1992.

[20] 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.

[21] V. Adve, V. V. Lam, and B. 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.

[22] I. Ben-Shaul, O. Holder, and B. Lavva, "Dynamic adaptation and deployment of distributed components in Hadas," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 27, no. 9, pp. 769–787, 2001.

[23] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. 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)*, (Vienna, Austria), July 2002.

[24] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[25] V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, pp. 95–106, September 2002.

[26] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, June 2002.

[27] M. Golm and J. Kleinöder, "Jumping to the meta level: Behavioral reflection can be fast and flexible," *Lecture Notes in Computer Science*, vol. 1616, 1999.

[28] A. Oliva and L. E. Buzato, "The implementation of Guaraná on Java," Tech. Rep. IC-98-32, Institute of Computing of the State University of Campinas, Sept. 1998.

[29] A. Popovici, T. Gross, and G. Alonso, "Dynamic homogenous AOP with PROSE," tech. rep., Department of Computer Science, Federal Institute of Technology, Zurich, 2001.

[30] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, pp. 294–324, April 1998.

[31] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "Towards highly configurable real-time object request brokers," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, April - May 2002.

[32] 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, Nov. 2001.

[33] S. M. Sadjadi and P. K. McKinley, "ACT: An adaptive CORBA template to support unanticipated adaptation," Tech. Rep. MSU-CSE-03-22, Department of Computer Science, Michigan State University, East Lansing, Michigan, August 2003. submitted to ICDCS 2004.

[34] 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.

[35] P. Tarr and H. Ossher, eds., *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (W17)*, May 2001.

[36] K. Czarnecki and U. Eisenecker, *Generative programming*. Addison Wesley, 2000.

[37] H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software," *Communications of the ACM*, vol. 44, no. 10, pp. 43–50, 2001.

[38] K. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-oriented programming with adaptive methods," *Communications of the ACM*, vol. 44, no. 10, pp. 39–41, 2001.

[39] 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*, pp. 1–24, September 2001.

[40] I. Welch and R. J. Stroud, "Kava - A Reflective Java Based on Bytecode Rewriting," in *Reflection and Software Engineering* (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), Lecture Notes in Computer Science 1826, pp. 157–169, Heidelberg, Germany: Springer-Verlag, June 2000.

[41] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of ACM*, pp. 51–57, October 2001.

[42] P. C. David, T. Ledoux, and N. 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.

[43] A. J. McAuley, "Reliable broadband communication using a burst erasure correcting code,," in *Proceedings of ACM SIGCOMM '90; (Special Issue Computer Communication Review)*, pp. 297–306, 1990.

[44] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, pp. 24–36, Apr. 1997.

[45] Swarmcast, "Release notes for Java FEC v0.5." http://www.swarmcast.com, 2001.