Architecture and Operation of an Adaptable Communication Substrate [*]

S. M. Sadjadi, P. K. McKinley, and E. P. Kasten
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{sadjadis,mckinley,kasten}@cse.msu.edu

Abstract

This paper describes the internal architecture and operation of an adaptable communication component called the MetaSocket. MetaSockets are created using Adaptive Java, a reflective extension to Java that enables a component's internal architecture and behavior to be adapted at run time in response to external stimuli. This paper describes how adaptive behavior is implemented in MetaSockets, as well as how MetaSockets interact with other adaptive components, such as decision makers and event mediators. Results of experiments on a mobile computing testbed demonstrate how MetaSockets respond to dynamic wireless channel conditions in order to improve the quality of interactive audio streams delivered to iPAQ handheld computers.

1 Introduction

The large-scale deployment of wireless communication services and advances in handheld computing devices enable users to interact with one another from virtually any location. Example applications include computer-supported cooperative work, management of large industrial sites, and military command and control environments. Such interactive distributed applications are particularly sensitive to the heterogeneity of the devices and networks used by the participants. Specifically, an application must accommodate devices, from workstations to PDAs, with widely varying display characteristics and system resource constraints. Moreover, the application must tolerate the highly dynamic channel conditions that arise as the user moves about the environment. One of the key challenges in designing future interactive systems is how to enable them to adapt the communication substrate to theses conditions at run time. Developing and maintaining such software is a nontrivial task. In this paper, we demonstrate the effectiveness of programming language support for the development and maintenance of the underlying communication infrastructure that must adapt to its environment.

Adaptability can be implemented in different parts of the system. One approach is to introduce a layer of adaptive middleware between applications and underlying transport services [1–3]. We are currently conducting an ONR-sponsored project called *RAPIDware* that addresses the design of adaptive middleware for dynamic, heterogeneous environments. Such systems require run-time adaptation, including the ability to modify and replace components, in order to survive hardware component failures, network outages, and security attacks.

A major focus of our study is on programming language support for adaptability. We previously developed Adaptive Java [4], an extension to Java that supports dynamic reconfiguration of software components. This paper focuses on an Adaptive Java component called the *MetaSocket* (for "metamorphic" socket). Although the socket abstraction is relatively low-level compared by many current distributed computing platforms (e.g., CORBA, Java RMI, and DCOM), its ubiquity in distributed applications, as well as in middleware platforms, makes it a good place to begin our studies. MetaSockets are created from existing Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli. In this paper, we focus on the *internal architecture and the operation* of MetaSockets and present a case study in the use of MetaSockets to support audio streaming over wireless channels. The case study, in which iPAQ handheld computers are used as audio "communicators," illustrates how MetaSockets interact with other adaptive components, such as decision makers and event mediators, to realize run-time adaptability in real-time communication services. The main contribution of this work is to propose a language-based approach to run-time adaptability and, through the case study,

[*] 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 CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724.

to reveal several subtle design issues that need to be addressed in the development of such software.

The remainder of this paper is organized as follows. Section 2 provides background information on the Adaptive Java programming language. In Section 3, we describe the design and implementation of a MetaSocket variation that is based on the Java MulticastSocket class. Section 4 discusses the case study in the the use of MetaSockets to support adaptive error control on wireless audio channels. Section 5 presents results of experiments that demonstrate the effectiveness of the proposed methods in adapting to dynamic changes in packet loss rate. Section 6 discusses related work, and Section 7 presents our conclusions and discusses future directions.

2 Adaptive Java Background

Adaptive Java [4] is an extension to Java that adds behavioral reflection to Java's structural reflection, by introducing new language constructs. These constructs are rooted in computational reflection [5,6], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. The basic building blocks used in an Adaptive Java program are *components*, which in this context can be equated to adaptable classes. The key programming concept in Adaptive Java is to provide three separate component interfaces: one for performing normal imperative operations on the object (*computation*), one for observing internal behavior (*introspection*), and one for changing internal behavior (*intercession*). Operations in the computation dimension are referred to as *invocations*. Operations in the introspection dimension are called *refractions*; they offer a partial view of internal structure and behavior, but are not allowed to change the state or behavior of the component. Operations in the intercession dimension are called *transmutations*; they are used to modify the computational behavior of the component.

An existing Java class can be converted into an adaptable component in two steps. In the first step, a *base-level* Adaptive Java component is constructed from the Java class through an operation called *absorption*. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added and removed from existing components at run time using meta-level transmutations. In the second step, *metafication* enables the creation of refractions and transmutations that operate on the base component. The meta-level can also be given a meta-level (meta-meta-level), which can be used to refract and transmute the meta-level. In theory, this reification of meta-levels for other meta-levels could continue indefinitely [6].

Adaptive Java Version 1.0 [4] is implemented using CUP [7], a parser generator for Java. CUP takes the grammar productions for the Adaptive Java extensions and generates an LALR parser, called ajc, which performs a source-to-source conversion of Adaptive Java code into Java code. Semantic routines were added to this parser such that the generated Java code could then be compiled using a standard Java compiler.

3 MetaSocket Design and Implementation

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

Figure 1 shows the absorption of a Java MulticastSocket base-level class by a SendMSocket base-level component, and the metafication of this component to a MetaSendMSocket meta-level component. Figure 1(a) depicts a Java MulticastSocket class and a subset of its public methods: receive(), send(), close(), joinGroup(), and leaveGroup(). Figure 1(b) shows a SendMSocket component, which is designed to be used as a *send-only* multicast socket. The SendMSocket component *absorbs* the Java MulticastSocket class and implements send() and close() invocations that can be used by other components. Other methods of the base-level class are occluded. A link between an invocation and a method indicates a dependency. For example, the send() invocation depends on the send() method, because its implementation calls that method. Figure 1(c) shows a MetaSendMSocket component, which metafies an instance of the SendMSocket component and provides a refraction, getStatus(), and two transmutations, insertFilter() and removeFilter(). The use and operation of these primitives will be explained shortly. Again, a link between a refraction (or transmutation) and an invocation indicates a dependency.

In a similar manner, a *receive-only* MetaSocket can be created for use on the receiving side of a communication channel. The RecvMSocket base-level component absorbs a Java MulticastSocket class. In addition to the receive() and close() invocations, this component also provides joinGroup() and leaveGroup() invocations, which are needed for joining and leaving an IP multicast group. All these invocations depend on their respective counterparts in the Java MulticastSocket class. The MetaRecvMSocket metafies an instance of RecvMSocket component and provides the same refractions and transmutations as does the MetaSendMSocket component. The code for MetaSendMSocket and MetaRecvMSocket can be loaded at run time, using the Java Class class and Java reflection package. This dynamic loading of adaptive code implies the ability of Adaptive Java applications to adapt to unanticipated changes at run time.

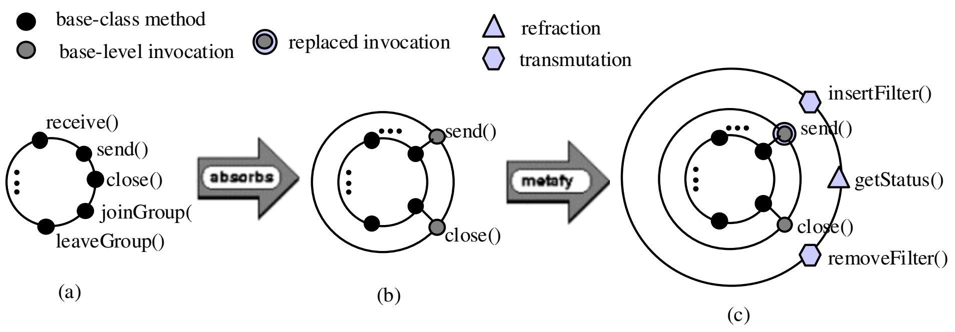

Figure 1: MetaSocket absorption and metafication: (a) Java MulticastSocket as the base-level class; (b) SendMSocket as the base-level component; (c) MetaSendMSocket, a filter-oriented meta-level component.

3.1 Internal Architecture and Operation

Figure 2 illustrates the internal architecture of both a MetaSendMSocket and a MetaRecvMSocket, as configured in our study. In this metafication, packets are passed through a pipeline of Filter components, each of which processes the packet. Example filter services include: auditing traffic and usage patterns, transcoding data streams into lower-bandwidth versions, scanning for viruses, and implementing forward error correction (FEC) to make data streams more resilient to packet loss. In some cases, such as auditing, a filter can act alone on either the sending or the receiving side of the channel. In other cases, such as FEC, modification of the packet stream introduced by a filter on the sender must be reversed by a peer filter on the receiver. In our implementation, when a packet is processed by a filter, an application-level header is prepended to the packet. On the receiver, these headers identify the processing order and filters required to reverse the transformations applied by the sender.

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

Sender Operation. Let us consider the sender, as shown in Figure 2(a). At the time of metafication, a SendMSocket component is encapsulated by the MetaSendMSocket component. Among other actions, the send() invocation of SendMSocket is replaced by a new send() invocation defined by the meta-level component. After metafication, any call to the base-level send() invocation is delegated to the meta-level send() invocation. This invocation adds a *terminator header* to the datagram packet it receives, which identifies packets that are ready for delivery to the application by the receiver. Next, the meta-level send() invocation stores this packet in firstPacketBuffer (the first packet buffer of the pipeline). Initially, both firstPacketBuffer and lastPacketBuffer refer to the same packet buffer. While lastPacketBuffer may change as new filters are inserted, always pointing to the last packet buffer in the pipeline, firstPacketBuffer remains fixed. When SendMSocket is metafied by MetaSendMSocket, a thread is created and assigned to the SendMSocket send() invocation. This thread loops, retrieving a packet from lastPacketBuffer, creating a datagram packet, and passing it to the original base-level send() invocation, which in turn transmits the packet to the multicast group using the send() method of the underlying MulticastSocket base class.

Receiver Operation. On the receiver, as shown in Figure 2(b), a MetaRecvMSocket encapsulates a base-level RecvMSocket component. The receiver can be added to the multicast group, either before or after metafication, by calling its joinGroup() invocation. Once metafied, a thread is assigned to the RecvMSocket receive() invocation. The thread loops continuously, calling receive() and placing the returned packet in firstPacketBuffer. The order of filters on the receiver is the mirror image of that on the sender with function inverted. Each filter in the pipeline processes a packet from its source packet buffer and places it in its destination packet buffer. Similar to the send() invocation on the sender, metafication replaces the base-level receive() invocation with the meta-level receive() invocation defined by MetaRecvMSocket. Instead of calling the RecvMSocket receive() invocation, the MetaRecvMSocket receive() invocation retrieves packets directly from lastPacketBuffer. Before returning the packet to the caller, however, the receive() invocation checks the packet's MetaSocket header. If a terminator header is found at the beginning of the packet, then receive() removes this header and returns the original packet to the caller. Otherwise, additional filter processing needs to be performed on the packet before delivering it to the application. In this case, receive() generates a *FilterMis-*

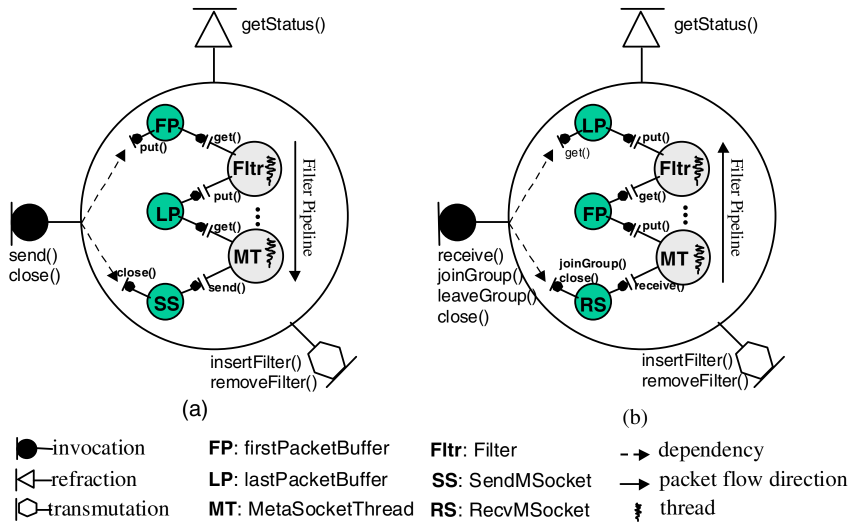

Figure 2: MetaSocket internal architecture: (a) MetaSendMSocket, a send-only metamorphic multicast socket; (b) MetaRecvMSocket, a receive-only metamorphic multicast socket.

matchEvent event containing the packet and the position of the required Filter in the filter pipeline. (Every Filter at the receiving side performs a similar task and compares the filter ID of the next packet to its ID.) This event is sent to the *EventMediator*, a singleton component in each addresss space that decouples event generators from event listeners [8]. The receive() invocation waits until the event has been handled, meaning that the needed filter has been inserted in the pipeline using the insertFilter() transmutation. Additional details on event handling are discussed in the next section.

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

3.2 Syntax of Absorption and Metafication

Figure 3.2 shows simplified Adaptive Java code for the SendMSocket component. A constructor is defined for this component that creates a new MulticastSocket and sets it as the base-level object for this component. Note that the base-level object is treated as a secret of the base-level component. A component that uses the SendMSocket component does not necessarily need to know anything about the underlying MulticastSocket or its interface. Two invocations, send() and close() are defined, but they simply call their associated methods from the base object. The code for RecvMSocket is similar. Once defined, SendMSocket and RecvMSocket can be used via their invocations.

```
public component SendMSocket
  absorbs java.net.MulticastSocket {
  /* constructor */
  public SendMSocket(...)  {
    setBase(new MulticastSocket(...));}
  /* invocations */
  public invocation void send(...)  {
    base.send(...); }
  public invocation void close() {
    base.close(); }
}
```

Figure 3: Excerpted code for SendMSocket.

The metafication of these base-level components can be defined at development time or later, at run time. Simplified code for MetaSendMSocket is shown in Figure 3.2. At any point during the execution of the application, a running SendMSocket component can be metafied by calling its constructor. The instance of SendMSocket passed to the constructor of this meta-component is designated as the base-level component. As described earlier, in addition to refractions and transmutations, an invocation, send(), is redefined in this meta-level component. Defining an invocation at the meta-level is used to replace an invocation of the base-level component. In this example, the new invocation does not call the Java MulticastSocket send() method. Instead, it places the packet in firstPacketBuffer defined as a private field of this meta-component. Another private field, filterPipeline, is an instance of `java.util.Vector` and keeps track of all the filters currently configured in the MetaSendMSocket. The refraction getStatus() returns a byte array containing the IDs of these filters. The transmutations insertFilter() and removeFilter() are used to insert and remove filters at specified positions in the filter pipeline. The code for MetaRecvMSocket is similar to that of MetaSendMSocket. In this case, however, the receive() invocation is redefined in the meta-level. In the new definition of this invocation, a packet from the lastPacketBuffer , if available, is delivered to the caller.

```
public component MetaSendMSocket
 metafy SendMSocket {
 /* constructor */
 public MetaSendMSocket(SendMSocket s)
  { setBase(s); }
 /* replacing the SendMSocket.send() */
 public invocation void send(...)  {...
  firstPacketBuffer.put(packet); ...}
 /* refractions */
 public refraction byte[] getStatus() {
  return filterPipeline.getStatus(); }
 /* transmutations */
 public transmutation void
  insertFilter(int pos, Filter f) {...
  filterPipeline.add(pos, f); ...}
 public transmutation Filter
  removeFilter(int pos) {...
  return filterPipeline.remove(pos); }
 /* private fields */
 private Vector filterPipeline =
  new Vector();
 PacketBuffer firstPacketBuffer =
  new PacketBuffer();
}
```

Figure 4: Excerpted code for MetaSendMSocket.

4 Adaptive Functionality in MetaSockets

The Java MulticastSocket class is used in many distributed applications. The MetaSockets described in the previous section provide the same imperative functionality to applications and can be used in place of regular Java sockets. In this section, we use an example Adaptive Java application to demonstrate how MetaSockets can further provide adaptive functionality by interacting with other supporting components, such as decision makers and event mediators. A key concept in this approach is that the adaptive functionality, whether it be related to quality-of-service, fault tolerance, or security, is not tangled with the application code. Rather, the "base" application code uses only invocations provided by MetaSockets, while the code that manipulates the behavior of MetaSockets is localized. This *separation of concerns* [9] depicted in Figure 5, leads to code that is easier to maintain and evolve to incorporate new adaptive functionality. In the following example, we use MetaSockets to support adaptable quality-of-service by reacting to changes in the quality of the wireless channel.

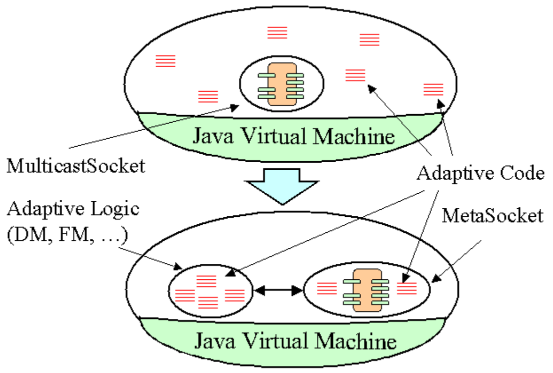

Figure 5: Example of separation of concerns using MetaSockets.

4.1 ASA Architecture and Operation

In this study, we modified an audio streaming application (ASA) to use MetaSockets instead of regular Java sockets, and we added components to manage the adaptive behavior. We then experimented with the ASA by streaming live audio from a desktop workstation to multiple iPAQ handheld computers over an 802.11b wireless local area network (WLAN). The experimental configuration is depicted in Figure 6.

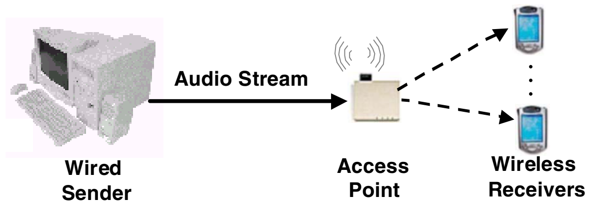

Figure 6: Physical experimental configuration.

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

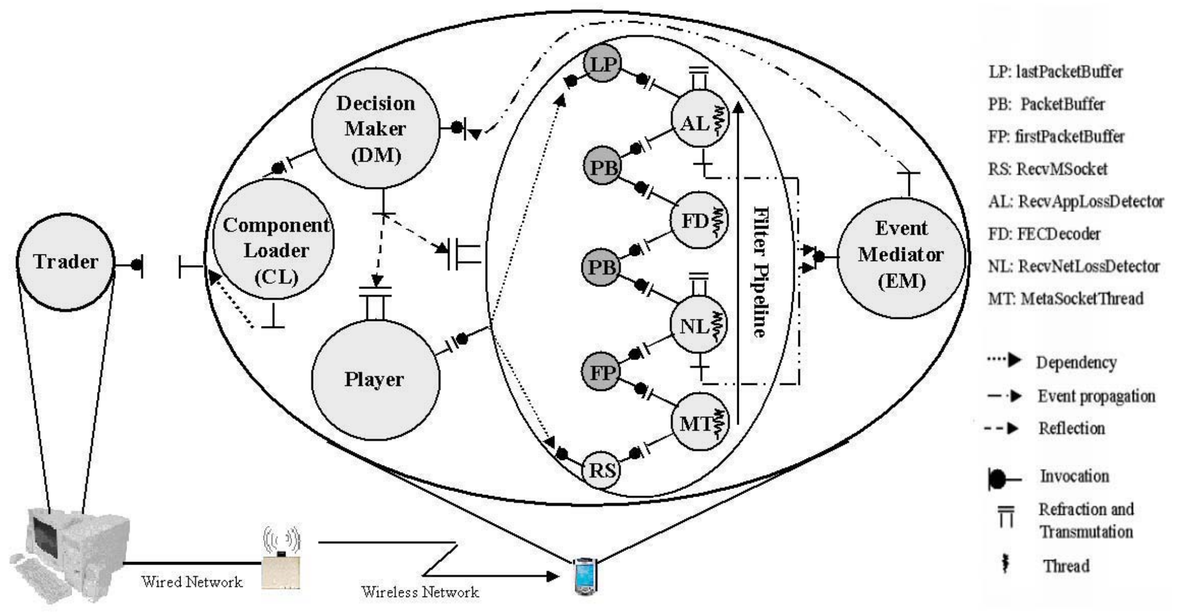

Figure 7: Interaction among components in the Audio Streaming Application.

receives the audio data using the receive() invocation of a MetaSocket and plays the data using the `javax.sound` package.

Figure 7 illustrates the major parts of the receiving side of the ASA; the sending side has a similar structure. Most of the system executes on an iPAQ handheld computer, but one program, called a *Trader*, executes on a desktop workstation. The two systems communicate over the WLAN. In Adaptive Java, each address space comprises one or more components, each of which in turn may comprise several interacting components. The program running on the iPAQ in Figure 7 comprises five main components: a Player, a DecisionMaker, an EventMediator, a ComponentLoader, and a MetaRecvMSocket. The MetaRecvMSocket contains several components that together implement the filter pipeline. As indicated, some of these components are metafied and therefore offer refractive and transmutative interfaces, whereas others are simple base-level components that offer only invocations to other components. The flow of events among components, via an EventMediator, is also shown.

A DecisionMaker (DM) is an optional subcomponent within any Adaptive Java component. According to a set of rules applied to the current situation, a DM controls all of the nonfunctional behavior of the subcomponents of its container component. DMs are arranged hierarchically, such that a DM inherits rules from a higher-level DM and might provide rules to lower-level DMs. (In our simple example application, the main component on the iPAQ contains a single DM.) Depending on its rules and the current situation, a DM might decide to metafy or change the configuration of an existing component by invoking transmutations of the component. A transmutation might simply set the value of an internal variable, or might involve the insertion or removal of a subcomponent (such as a filter, in our example). In the insertion case, the DM contacts the ComponentLoader (CL) and requests the needed component. The CL is unique to an address space. If the CL does not find the component in its cache, it sends a request to a component *Trader*, which may reside on another computing system. The Trader returns a component implementation corresponding to a syntactic or semantic component request. In our current implementation, we use simple identifiers to search for components. Eventually, the CL uses the `java.lang.ClassLoader` to load this implementation, creates an instance of this class, and returns it to the local DM. The ability to dynamically load components is especially important for mobile devices, where resources might be limited and overhead should be minimized.

Components can interact directly via invocations, refractions and transmutations. To support asynchronous interactions, we implemented an event service. An EventMediator (EM) decouples event generators from event *listeners* [8]. The ASA sender and receiver each contain a single EM that handles all events in the respective program. A listener registers its interest in an event by calling the EM's registerInterest() invocation. When an event is detected by a component, it calls the notify() invocation of the EM. The EM records the event and subsequently alerts all listeners by calling their notify() invocations. To complete the earlier discussion on missing filters, let us consider the situation in which the thread in the receive() meta-level invocation detects that another filter needs to be configured in the pipeline. A *FilterMismatchEvent* event is sent to the EM, which forwards it to the DM. The DM decides to insert a new filter based on information carried by the event and the pipeline status retrieved using the getStatus() refraction. The DM requests the CL to load the missing filter, after which the DM inserts it at the proper location in the pipeline.

4.2 Filter Components

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

FEC is widely used in wireless networks. In wireless environments, factors such as signal strength, interference, and antennae alignment produce dynamic and location-dependent packet losses. In current wireless LANs, these problems affect multicast connections more than unicast connections, since the 802.11b MAC layer does not provide link-level acknowledgements for multicast frames. FEC can be used to improve reliability by introducing redundancy into the data channel. Our filters use (n, k) block erasure codes [10]. As shown in Figure 8, k source packets are converted into a group of n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [10]. These codes are ideal for wireless multicasting, since a single set of parity packets can correct different packet losses among receivers.

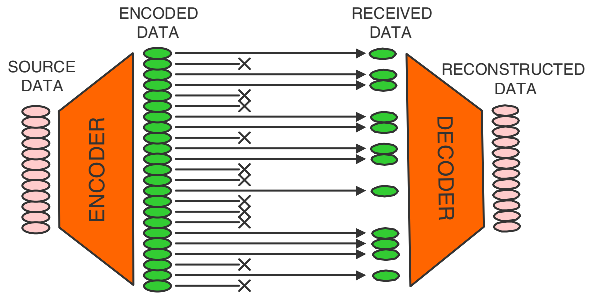

Figure 8: Operation of block erasure code.

The FECEncoder and FECDecoder components are extended from the Filter component and use a Java FEC package . The FECEncoder runs on the sender. This component retrieves k packets from its source packet buffer, generates $n - k$ parity packets, and places the original k packets plus the $n - k$ parity packets into its destination packet buffer. The FECDecoder runs at the receiving side and retrieves up to k packets from its source packet buffer, decodes them if possible, and places the recovered original k packets in its destination packet buffer. Any unneeded parity packets are simply dropped. If fewer than k out of the n packets arrive, for a given FEC group, then the FECDecoder retrieves any data packets and places them into its destination packet buffer. The MetaFECEncoder and MetaFECDecoder, shown in Figure 9, metafy the FECEncoder and FECDecoder components, respectively. Each provides a getNK() refraction and setNK() transmutation, which are used at run time to read and set the values of n and k. If a packet arrives with a different n or k value than is expected, the MetaFECDecoder fires a *FECMismatchNKEvent* event. In response, the DM uses setNK() transmutation and adjusts the values for k and n appropriately.

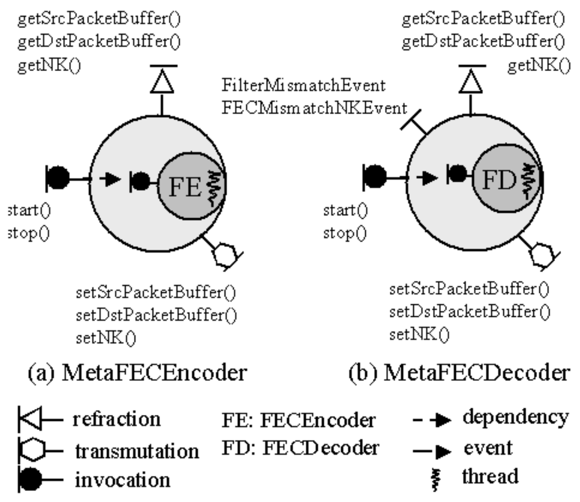

Figure 9: Design of forward error correction filters.

The second type of filter used in our case study, monitors events related to packet loss rate and reports these to the DM. We developed two sets of filters. The SendNetLossDetector and RecvNetLossDetector filters monitor the raw loss rate of the wireless channel. The SendAppLossDetector and RecvAppLossDetector filters monitor the packet loss rate as observed by the application, which may be lower than the raw packet loss rate due to the use of FEC. The metafied versions of these filters is shown in Figure 10. In our experiments, SendAppLossDetector is used as the first filter on the sender side, and RecvAppLossDetector is used as the last filter on the receiver. Conversely, SendNetLossDetector is the last filter on the sender, and RecvNetLossDetector is the first filter on the receiver. The sender's filters simply prepare packets by prepending a header containing the identifier of the corresponding peer filter on the receiver. Each filter on the receiver uses sequence numbers to calculate the packet loss rate over a specified window in the packet stream and stores this information in a vector. Metafying these components provides refractions and transmutations to read the current loss rate and to set or change upper and lower thresholds with respect to the loss rate.

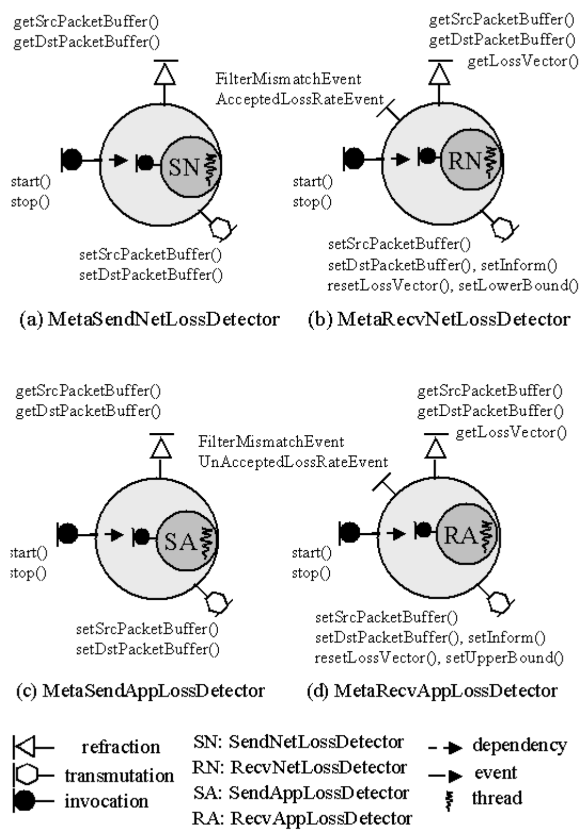

Figure 10: Design of packet loss monitoring filters.

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

5 Performance Evaluation

To evaluate the effect of MetaSockets on the performance of audio streaming, we conducted an experiment using the ASA. The Recorder program is configured to record 8000 samples per second of live audio, using a single channel at 8 bits per sample. Samples are collected into 128-byte packets packets, that is, each packet contains 16 milliseconds of audio data. We used $(8, 4)$ FEC filters. The upper threshold for the RecvAppLossDetector to generate an *UnAcceptableLossRateEvent* is 30%, and the lower threshold for the RecvNetLossDetector to generate an *AcceptableLossRateEvent* is 10%. One well-known difficulty in conducting experimental research in wireless environments is the ability to reproduce results, given the highly dynamic nature of the medium [11]. In these tests, we created artificial losses by dropping packets in software according to a predefined loss function. In this way, we are able to compare the effects of different parameter settings on the behavior of MetaSockets.

Figure 11 plots packet loss as observed by the two loss monitoring filters on the receiver. The Network Packet Loss curve experiences two periods of high packet loss. The Application Packet Loss curve shows the effect of dynamic insertion and removal of the FEC filter, according to the rules described in Section 4.2. When the program begins execution, the sender inserts a SendAppLossDetector filter into its MetaSocket, which quickly causes the receiver to insert the corresponding RecvAppLossDetector. At packet set 8 (meaning the 800th packet), the RecvAppLossDetector filter detects that the loss rate has passed the upper threshold. The filter fires an *UnAcceptableLossRateEvent*, causing the local DM to request an FEC filter. The global DM decides, based on its set of rules, to insert two filters, an FECEncoder filter with default parameters $n = 8$ and $k = 4$, and a SendNetLossDetector filter, at the second and third positions in the MetaSendMSocket filter pipeline, respectively. When packets containing the headers of the two new filters begin arriving at the receiver, the RecvAppLossDetector detects a packet header that does not match its own identifier. Therefore, it fires a *FilterMismatchEvent* at two different times, one for each new packet type. These events result in the insertion of a RecvNetLossDetector filter and a FECDecoder filter at the first and second positions in the MetaRecvMSocket filter pipeline, respectively.

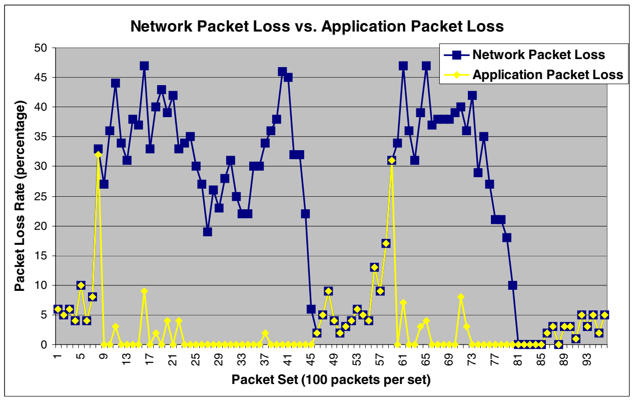

Figure 11: MetaSocket packet loss behavior .

As shown in Figure 11, the $(8,4)$ FEC code is very effective in reducing the packet loss rate as observed by the application from packet set 8 to packet set 45. which used FEC codes in an ad hoc proxy architecture. At packet set 45, the RecvNetLossDetector detects that the loss rate has dipped below the 10% lower threshold, so it fires an *AcceptableLossRateEvent*. In response, the local DM sends a request to the global DM to remove the FEC filter. The DM complies, since under low-loss conditions, the 100% overhead of an $(8,4)$ FEC code simply wastes bandwidth. It also removes the SendNetLossDetector filter in order to minimize data stream processing under favorable conditions. The arrival of packets without the two headers produces two *FilterMismatchEvent* events at the receiving side, and the peer filters are removed. As a result, the loss rate experienced by the application is again the same as the network loss rate. At packet set 60, the FEC filter is again inserted, due to high loss rate, and it is later removed at packet set 80. Considering Figure 11 as a whole, we see that the loss rate observed by the application is very low, with the exception of two brief spikes. In order to minimize overhead, FEC is applied only when necessary. This example illustrates how Adaptive Java components can interact at run time to recompose the system in response to changing conditions. While a task such as FEC filter management can be implemented in an ad hoc manner, run-time metafication in Adaptive Java enables such concerns to be added to the system after it is already deployed and executing.

6 Related Work

Several adaptive middleware projects involve adaptive extensions to CORBA [1–3, 12]. In contrast to a CORBA-based design, however, our focus in this study is on programming language constructs to support adaptive interfaces to arbitrary components. The MetaSocket architecture described in this paper bears some resemblance to portable interceptors, initially introduced in TAO [13, 14], standardized in CORBA 2.6, and used also in dynamicTAO [15]. However, Adaptive Java allows "interception" of any invocation, using a metafication that includes a new definition for the invocation, and in this sense is more general.

Other researchers have addressed the use of programming language constructs to realize adaptable behavior. For example, JDrums [16] and MetaJava [17] introduce a method for supporting dynamic update of Java programs; however, both require modifications to the JVM. Our "weaving" of adaptive code with the base application is reminiscent of aspect-oriented programming [9]. Although many projects in the AOP community address compile-time weaving [18], a growing number of projects focus on run-time composition [19, 20]. By defining a reflection-based component model, Adaptive Java also supports run-time reconfiguration but is not restricted to the AOP model, which requires identification of predefined "pointcuts" at compile time. A related concept is composition filters [21], which provide a mechanism for disentangling the cross-cutting concerns of a software system. Besides filters, however, Adaptive Java can be applied to components that interact in arbitrary ways.

The PCL project [22] also focuses on language support for run-time adaptability and is perhaps most closely related to our work. PCL is intended for use directly by applications. Our concept of "wrapping" classes with base components is similar to the use of *Adaptors* used in PCL. However, modification of the base class in PCL appears to be limited to changing variable values, whereas Adaptive Java transmutations can modify arbitrary structures or subcomponents. Moreover, by combining encapsulation with metafication, Adaptive Java can be used to realize adaptations in multiple meta-levels.

7 Conclusions and Future Directions

In this work, we investigated the use of of Adaptive Java to support run time adaptation in iPAQ handheld computers used as audio "communicators." Our study focused on an adaptable component called the MetaSocket. While we have discussed the use of MetaSockets previously [23], this paper is the first to describe the internal architecture and operation of MetaSockets. Specifically, we described in detail how adaptive behavior is implemented and how MetaSockets interact with other adaptive components, including decision makers and event mediators. Results from experiments on a mobile computing testbed demonstrate the effectiveness of these methods in responding to dynamic wireless channel conditions. It is our hope that the details of this design, combined with the case study, will be useful to other researchers and developers who are interested in language-supported, run-time adaptability for distributed object-oriented systems.

While this paper demonstrated the application of

MetaSockets to a specific communication service, we emphasize that the Adaptive Java mechanisms are general. Any component in the system can be metafied and adapted at run time. Currently, we are investigating the use of Adaptive Java to address other key areas where software adaptability is needed in distributed systems: dynamically changing the fault tolerance properties of components, adaptive security policies dynamically woven across components, mitigation of the heterogeneity of system display characteristics, and energy management strategies for battery-powered devices.

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`.

References

[1] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons, "The design and performance of a pluggable protocols framework for object request broker middleware," in *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, Massachusetts), August 1998.

[2] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. M. aes, 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.

[3] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken, "QuO's runtime support for quality of service in distributed objects," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, (The Lake District, England), September 1998.

[4] E. Kasten, P. K. McKinley, S. 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, Austia), July 2002.

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

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

[7] S. E. Hudson, ed., *CUP User's Manual.* Usability Center, Georgia Institute of Technology, July 1999.

[8] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic support for distributed applications," *IEEE Computer*, vol. 33, no. 3, pp. 68–76, 2000.

[9] 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.

[10] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, April 1997.

[11] D. A. Eckhardt and P. Steenkiste, "A trace-based evaluation of adaptive error correction for a wireless local area network," *Mobile Networks and Applications*, vol. 4, no. 4, pp. 273–287, 1999.

[12] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "A software architecture for adaptive distributed multimedia applications," *IEE Proceedings - Software*, vol. 145, no. 5, pp. 163–171, 1998.

[13] N. Wang, K. Parameswaran, and D. Schmidt, "The design and performance of meta-programming mechanisms for object request broker middleware," in *USENIX on Object - Oriented Technologies and Systems (COOTS)*, Jan/Feb 2001.

[14] D. C. Schmidt, "Middleware for real-time and embedded systems," *Communication of the ACM*, vol. 45, pp. 43–48, June 2002.

[15] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Communications of the ACM*, pp. 33–38, June 2002.

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

[17] M. Golm, "Design and implementation of a meta architecture for Java," Master's thesis, Friedrich-Alexander-University, Erlangen-Nurenburg, January 1997.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP*, pp. 327–353, 2001.

[19] E. Truyen, B. N. Jörgensen, W. Joosen, and P. Verbaeten, "Aspects for run-time component integration," in *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, (Sophia Antipolis and Cannes, France), 2000.

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

[21] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, vol. 44, pp. 51–57, October 2001.

[22] 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.

[23] P. K. McKinley, S. Sadjadi, E. P. Kasten, and R. Kalaskar, "Programming language support for adaptable wearable computing," in *Proceedings of the Sixth International Symposium on Wearable Computers*, (Seattle, Washington), October 2002.