MetaSockets: Run-Time Support for Adaptive Communication Services *

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

Rapid improvements in mobile computing devices and wireless networks promise to provide a foundation for ubiquitous computing. However, comparable advances are needed in the design of mobile computing applications and supporting middleware. Distributed software must be able to adapt to dynamic situations related to several cross-cutting concerns, including quality-of-service, fault-tolerance, energy management, and security. We previously introduced Adaptive Java, an extension to the Java programming language, which provides language constructs and compiler support for the development of adaptive software. This paper describes the use of Adaptive Java to develop an adaptable communication component called the MetaSocket. MetaSockets are created from existing Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli. MetaSockets can be used for several distributed computing tasks, including audits of traffic patterns for intrusion detection, adaptive error control on wireless networks, and dynamic energy management for handheld and wearable computers. This paper focuses on the internal architecture and operation of MetaSockets. We describe how their adaptive behavior is implemented using Adaptive Java programming language constructs, 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.

Keywords: mobile computing, computational reflection, adaptive middleware, real-time communication, event-based systems, error control, intrusion detection.

Submitted as a **Research Paper** to DOA 2002.

* 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.

1 Introduction

The large-scale deployment of wireless communication services and advances in handheld computing devices enable users to collaborate from virtually any location. Example applications include computer-supported cooperative work, management of large industrial sites, and military command and control environments. Given their synchronous and interactive nature, however, collaborative computing applications are particularly sensitive to the heterogeneity of the devices and networks used by 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. To enable effective collaboration among users, including the transfer of applications from one device to another, systems must adapt to these 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 distributed object-oriented applications that must adapt to their 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 [1–5]. Some researchers argue that an adaptive middleware framework should insulate application components from platform variations and changes in external conditions by encapsulating adaptive behavior in an underlying middleware substrate. However, others maintain that many *context-aware* applications are effective only when they can explicitly take advantage of the dynamics of the environment [6]. As a result, a number of frameworks have been proposed to assist the application developer in managing context and adapting to changing conditions [7–13].

Regardless of what parts of the system implement adaptive behavior, an important issue is how to develop adaptive software that can modify its behavior in ways not necessarily anticipated during the original software development, but required later in a deployed system. This problem is especially important to systems that must continue to operate correctly during exceptional situations. Examples include systems used to manage critical infrastructures, such as electric power grids, telecommunication systems, and nuclear facilities, as well as command and control systems. 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. We are currently conducting an ONR-sponsored project called *RAPIDware* that addresses the design of adaptive software to support critical infrastructure protection in dynamic, heterogeneous environments. The RAPIDware project complements many of the projects cited earlier by focusing on software technologies and programming abstractions for building adaptable systems. The techniques use rigorous software engineering principles to help preserve functional properties of the code as the system

adapts to its environment.

A major focus of our study is on programming language support for adaptability. We previously developed Adaptive Java [14], an extension to Java that supports dynamic reconfiguration of software components. This paper describes the use of Adaptive Java to develop an adaptable communication 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, EJB, .NET, and so forth), its ubiquity in distributed applications, as well as in those 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 the RAPIDware project, we are using MetaSockets for several distributed computing tasks, including component auditing for intrusion detection, adaptive error control on wireless networks, and dynamic energy management for handheld and wearable computers. In this paper, we focus on the internal architecture and the operation of MetaSockets and present a detailed 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 "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, 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 [14] language constructs are rooted in computational reflection [15, 16], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. Typically, the *base-level* functionality of the program is augmented with one or more *meta* levels, each of which observes and manipulates the base level. In object-oriented environments, the entities inhabiting a meta level are called meta-objects, and the collection of interfaces provided by a set of meta-objects is called a meta-object protocol, or MOP.

2.1 Adaptable Component Model

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. Refractions and transmutations embody limited adaptive logic and are intended for defining *how* the base level can be inspected and changed. The logic defining *why and when* these operations should be used is provided by other components that use meta-level operations (refractions and transmutations) to implement their decisions.

Implementing a three-dimensional interface to each Adaptive Java component is intended to address a key issue that arises in the use of reflection, namely, the degree to which the system should be able to change its own behavior [17]. A completely open implementation implies that an application can be recomposed entirely at run time, which may produce undesired behavior. On the other hand, limiting adaptability also limits the ability of the system to survive adverse situations. Hence, rather than considering MOPs as orthogonal portals into base-level functionality [18], we propose an alternative model in which MOPs are constructed from primitives, namely, refractions and transmutations. Figure 1 illustrates this concept. Different MOPs can be defined for different cross-cutting concerns: communication quality-of-service, fault tolerance, security, energy management, and so on. We argue that defining different MOPs in terms of a well-defined set of primitives controls the degree of adaptation and facilitates the coordination of their activities through components such as decision makers and event mediators.

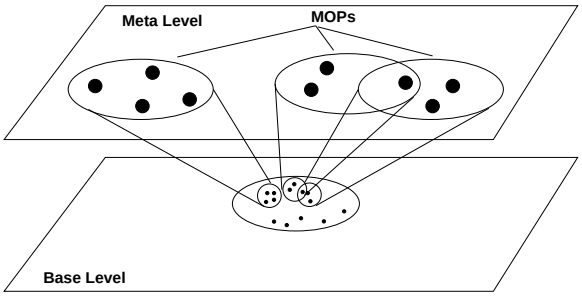


Figure 1. MOPs implemented with primitive operations.

An existing Java class is converted into an adaptable component in two steps, as shown in Figure 2. In the first step, a *base-level* Adaptive Java component is constructed from the Java class through an operation called *absorption*, which uses the `absorbs` keyword. As part of the absorption procedure, mutable methods

called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added and removed from existing components at run time using meta-level transmutations. We emphasize that the relationship between invocations on the base-level component and methods on the base-level class need not be one-to-one. Some of the base-level methods may be occluded or even combined under a single invocation. In this manner, the base-level component defines, explicitly, those parts of the original class are to be adaptable. For example, we might create a base-level socket component by absorbing a Java socket class. However, the base-level socket may provide a customized interface for use in a particular application domain, such as video or audio streaming.

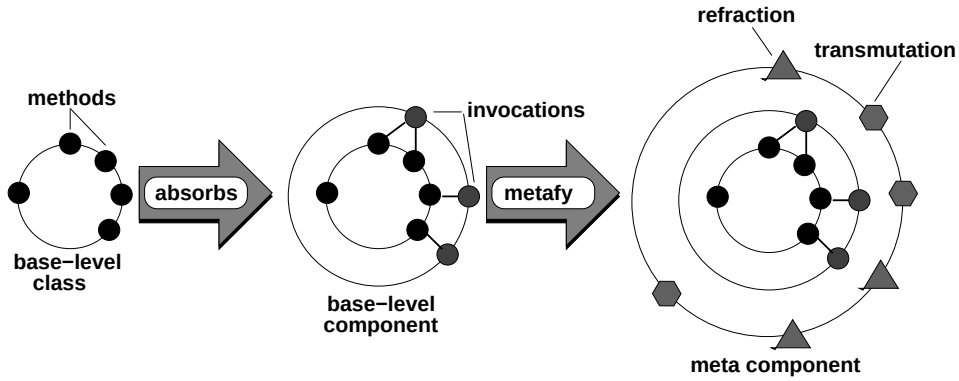


Figure 2. Component absorption and metafication.

In the second step, *metafication* enables the creation of refractions and transmutations that operate on the base component, as shown in Figure 2. Meta components are defined using the `metafy` keyword. Continuing our socket example, a transmutation might be defined to insert compression or encryption modules into a socket, while a refraction might be used to observe traffic patterns on behalf of an intrusion detection system. We emphasize that 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 infinitely [16].

2.2 Adaptive Java Implementation

We used CUP [19], a parser generator for Java, to implement Adaptive Java Version 1.0 [14], which is used in this study. CUP takes our 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.

Absorbed or metafied classes or components are encapsulated, or wrapped, within an outer component. The inner component is called the outer component's *base*. Invocations, defined by the outer component,

can be mapped to one or more invocations or methods of its base, or may extend a component's function by adding invocations that are not implemented by the base. In Adaptive Java 1.0, upon source-to-source compilation, Adaptive Java invocations, refractions and transmutations are converted into Java classes. Instantiation of a component also constructs a component's default invocations, and stores them in a hash map. Figure 2.2 depicts the Java code generated for the Adaptive Java invocation call: `rc = invoke instream.read(bytes)`. When an invocation is called, the invocation name (a Java String) is used as a key for locating the invocation object in a component's hash map. The invocation is cast, using a Java interface, such that the arguments and return types are those expected by the caller. Finally, the invocation object's invoke() method is called with parameters provided by the caller. When execution is complete, control is returned to the caller.

```
rc = ((read_role)(instream.
     __adaptj_invoke_invocation__("read"))).
     invoke(bytes);
```

Figure 3. Java code produced for an Adaptive Java invocation call.

Indirection is a key aspect of the above process. By storing invocations in a hash map, program objects do not hold references to invocations directly. That is, an invocation reference is retrieved only when an invocation is called, and relinquished once execution is complete. Moreover, invocations can be removed, exchanged or added as needed. As such, a component can be adapted by modifying its set of invocations. Moreover, refractions and transmutations can be defined to support the construction of, possibly overlapping, MOPs.

We emphasize that Adaptive Java is a prototype language whose purpose is to improve our understanding of the language constructs and mechanisms that are desirable in dynamic and adaptive languages. Eventually, we intend to reimplement Adaptive Java as a compiled language and extend the Java JVM to better support dynamic languages like Adaptive Java.

3 MetaSocket Design and Implementation

To explore the ability of Adaptive Java for support of run-time adaptability, we used this language to design and implement a "metamorphic" socket (MetaSocket) component. 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 4 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 4(a) depicts a Java MulticastSocket class and a subset of its public methods: receive(), send(), close(), joinGroup(), and leaveGroup(). Figure 4(b) shows a SendMSocket component, which is designed to be used as a *send-only* multicast socket. The SendMSocket component *absorbs* the Java multicast socket 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 4(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/transmutation and an invocation indicates a dependency.

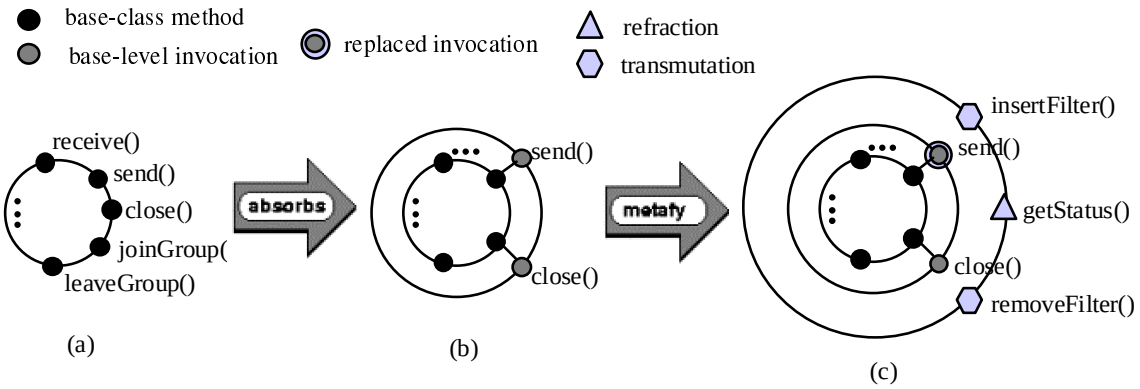


Figure 4. MetaSocket absorption and metafication: (a) Java MulticastSocket as the base-level class; (b) SendMSocket as the base-level component; (c) MetaSendMSocket, a filter-oriented meta-level component.

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.

3.1 Internal Architecture and Operation

Figure 5 illustrates the internal architecture of both a MetaSendMSocket and a MetaRecvMSocket, as configured in our study. (When adapting MetaSockets for other purposes, such as intrusion detection or energy management [20], the base-level component is metafied with different refractions and transmutations.) In this metafication, packets are passed through a pipeline of Filter components, each of which processes the packet. Example filter services include: auditing traffic and usage patterns, transcoding data streams into lower-bandwidth versions, scanning for viruses, and implementing forward error correction (FEC) to make data streams more resilient to packet loss. In some cases, such as auditing, a filter can act alone on either the sending or the receiving side of the channel. In other cases, such as FEC, modification of the packet stream introduced by a filter on the sender must be reversed by a peer filter on the receiver. In our implementation, when a packet is processed by a filter, an application-level header is prepended to the packet. On the receiver, these headers identify the processing order and filters required to reverse the transformations applied by the sender.

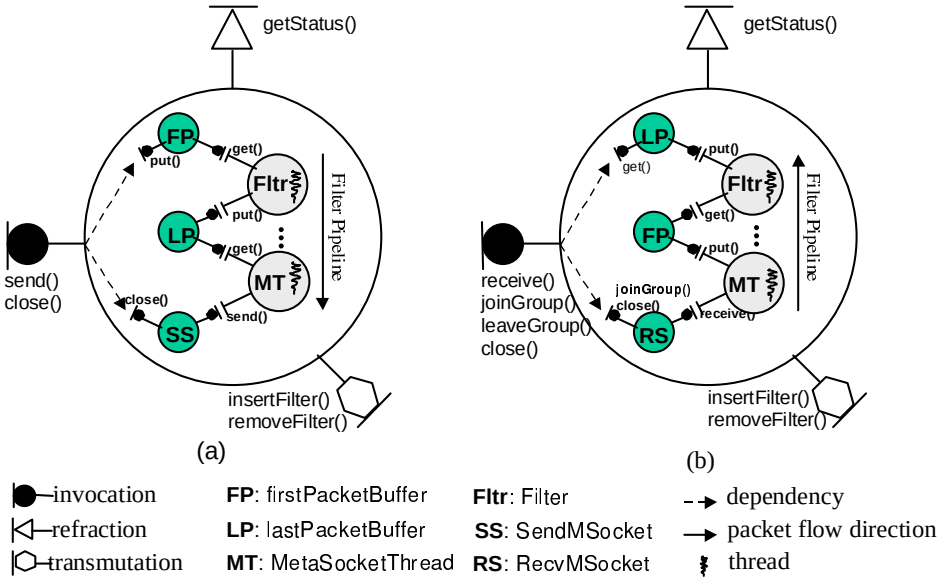


Figure 5. MetaSocket internal architecture: (a) MetaSendMSocket, a send-only metamorphic multicast socket; (b) MetaRecvMSocket, a receive-only metamorphic multicast socket.

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

in the pipeline is the source packet buffer of the next filter.

Sender Operation. Let us consider the sender, as shown in Figure 5(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 the first packet buffer, firstPacketBuffer, 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 the lastPacketBuffer, creating a datagram packet, and passing it to the original base-level send() invocation, which 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 5(b), a MetaRecvMSocket encapsulates a base-level RecvMSocket component. The receiver can be added to the multicast group, either before or after metafication, by calling its joinGroup() invocation. Once metafied, a thread is assigned to the RecvMSocket receive() invocation. The thread loops continuously, calling receive() and placing the returned packet in firstPacketBuffer. The order of filters on the receiver is the mirror image of that on the sender with function inverted. Each filter in the pipeline processes a packet from its source packet buffer and places it in its destination packet buffer. Similar to the send() invocation on the sender, metafication replaces the base-level receive() invocation with the meta-level receive() invocation defined by MetaRecvMSocket. Instead of calling the RecvMSocket receive() invocation, the MetaRecvMSocket receive() invocation retrieves packets directly from lastPacketBuffer. Before returning the packet to the caller, however, the receive() invocation checks the packet's MetaSocket header. If a terminator header is found at the beginning of the packet, then receive() removes this header and returns the original packet to the caller. Otherwise, additional filter processing needs to be performed on the packet before delivering it to the application. In this case, receive() generates a *FilterMismatchEvent* event containing the packet and the position of the required Filter in the filter pipeline. (Actually, 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. 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. More 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 (using the filter's setSrcPacketBuffer() transmutation), and starts the new filter (using the filter's start() invocation). The removeFilter() stops the filter that should be removed (using stop() invocation), flushes all the packets out of this filter's destination packet buffer (using flush() invocation), destroys this filter (using setDestroy(true) transmutation), 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 the list of all IDs of filters currently configured in the pipeline.

3.2 Syntax of Absorption and Metafication

Figure 6 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 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 java.net.MulticastSocket(...)); }

    /* invocations */
    public invocation void send(...) { base.send(...); }
    public invocation void close() { base.close(); }
}
```

Figure 6. Excerpted and simplified 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 7. 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 a packet buffer defined as a private field of this meta-component. Another private field, filterPipeline, is an instance of `java.util.Vector` for keeping 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.

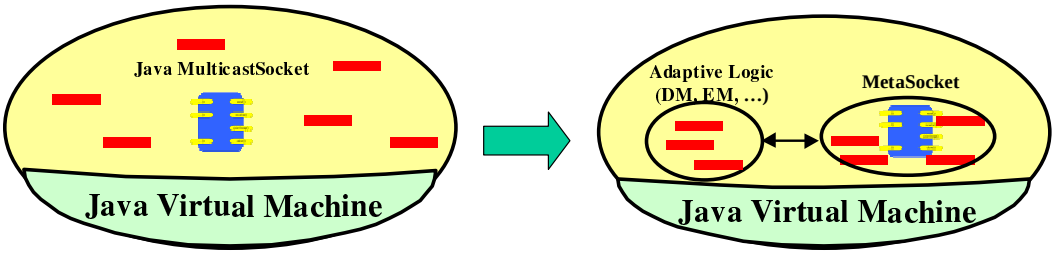
```
public component MetaSendMSocket metafy SendMSocket {
    /* constructor */
    public MetaSendMSocket(SendMSocket sendMSocket)
        { setBase(sendMSocket); }

    /* invocation that "replaces" the send() invocation in SendMSocket */
    public invocation void send(...)
        { ...  firstPacketBuffer.put(packet); ... }

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

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

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

Figure 7. Excerpted and simplified code for MetaSendMSocket.

4 Implementing Adaptive Functionality with 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* [21] depicted in Figure 8, 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.



(a) (b)

Figure 8. Example of separation of concerns using MetaSockets: (a) adaptive code scattered across the application code; (b) adaptive code factored into meta components and controlling logic.

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 added components to manage the adaptive behavior. We then experimented with the ASA, 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 9. 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 [22]. On each receiving station, the *Player* receives the audio data using the receive() invocation of a MetaSocket and plays the data using the `javax.sound` package.

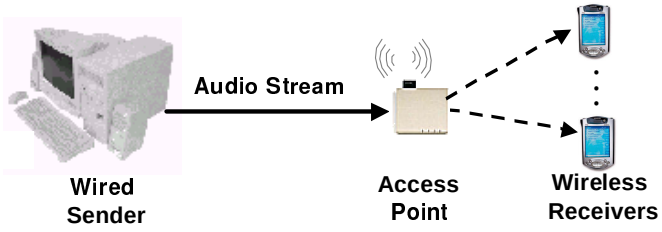


Figure 9. Physical experimental configuration.

Figure 10 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 10 comprises five main components: the 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, while 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.

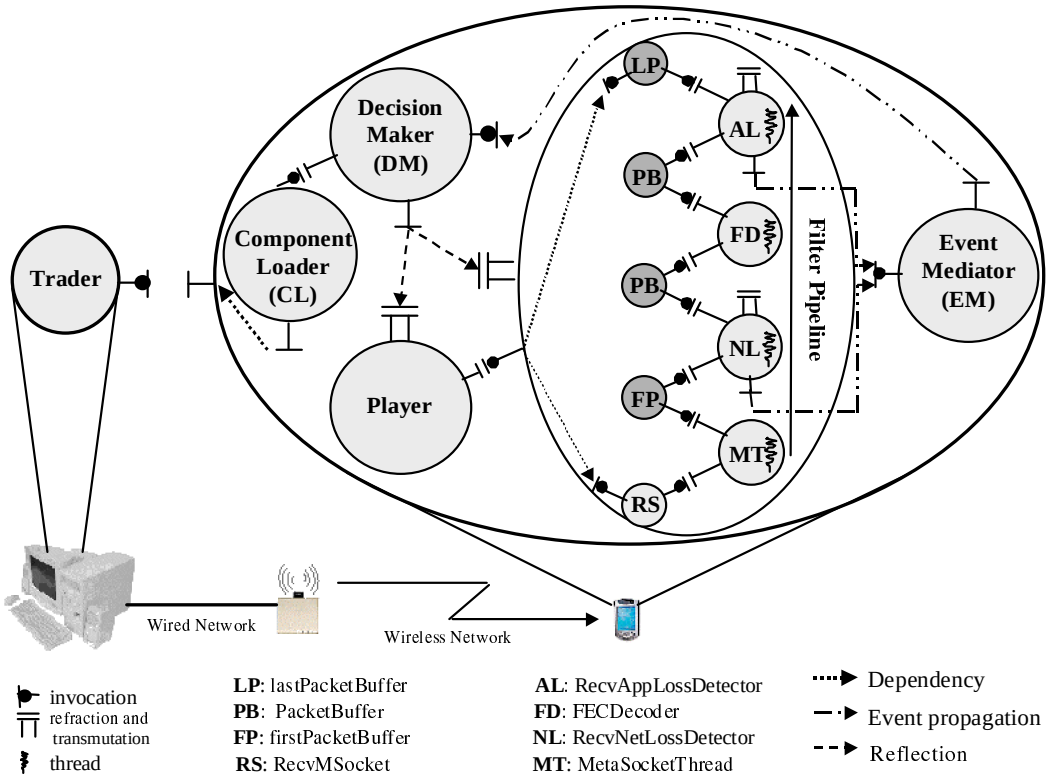


Figure 10. Interaction among components in the Audio Streaming Application.

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 one of its transmutations. 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* [23]. 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 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 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 [24]. 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 [25]. As shown in Figure 11, 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 [25]. These codes are ideal for wireless multicasting, since a single set of parity packets can correct different packet losses among receivers.

The FECEncoder and FECDecoder components are extended from the Filter component and use a public

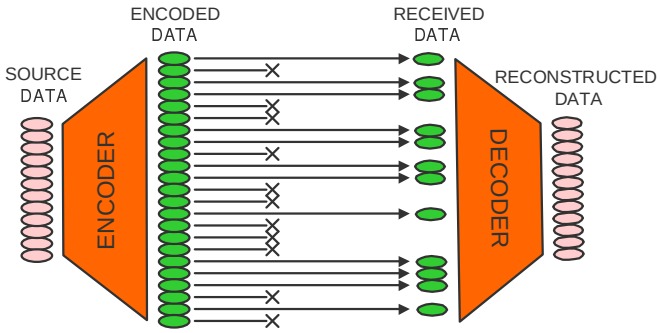
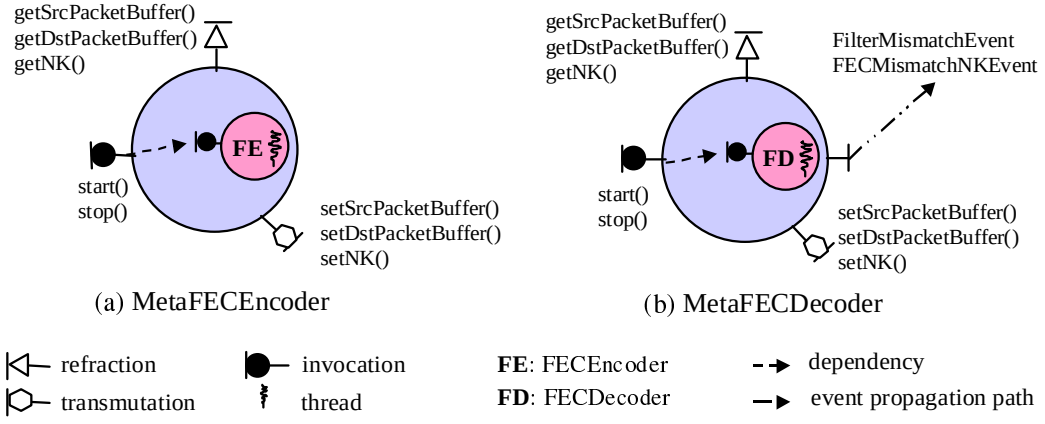


Figure 11. Operation of block erasure code.

domain Java FEC package [26]. 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 of the application. It 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 12, 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.



(a) MetaFECEncoder (b) MetaFECDecoder

Figure 12. 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 RecvAppLossDe-

tector 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 13. 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. Metafing 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.

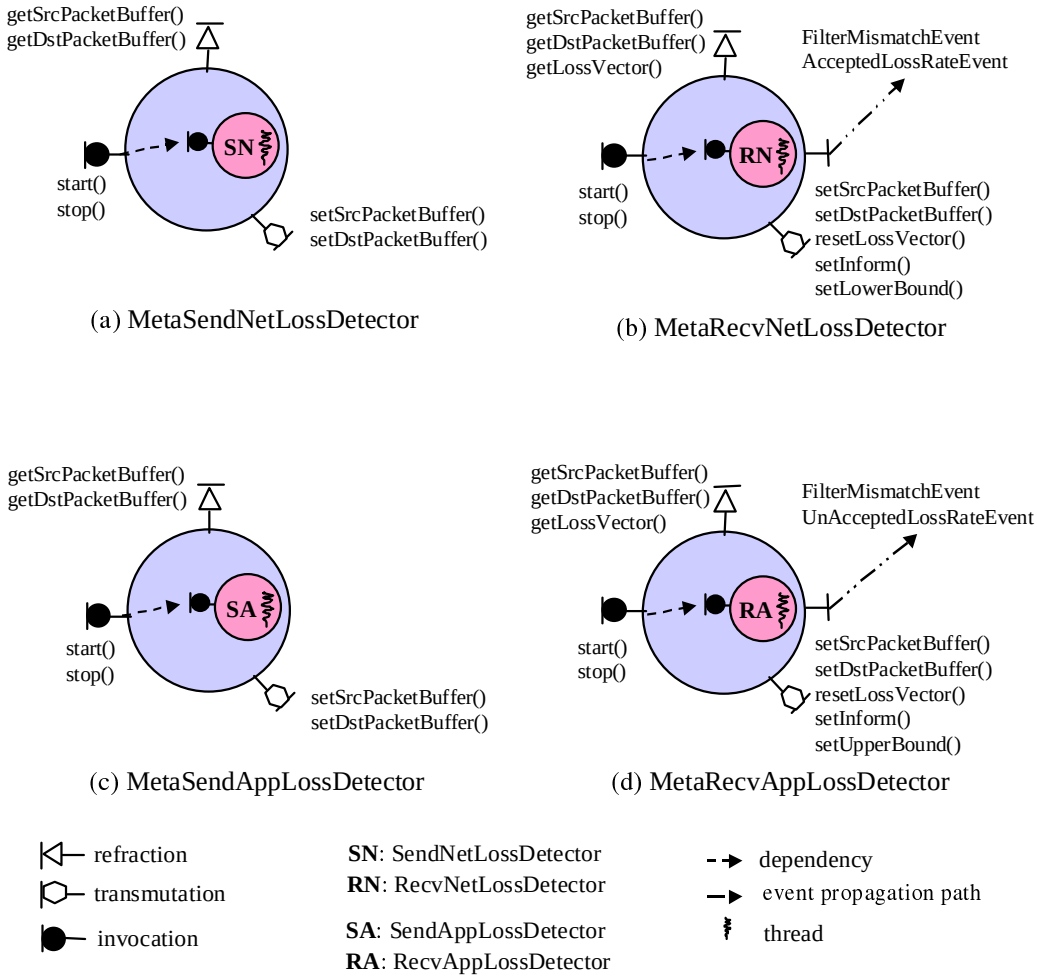


Figure 13. Design of packet loss monitoring filters.

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 a 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 [27]. 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 14 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. 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. 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. In fact, 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.

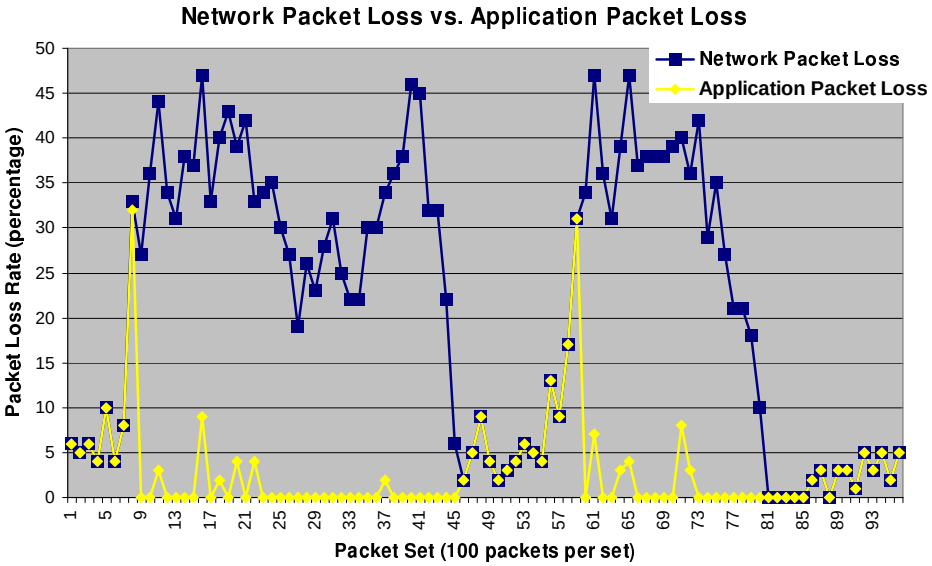


Figure 14. MetaSocket packet loss behavior will dynamic filter insertion.

As shown in Figure 14, 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. This result is consistent with our earlier studies [22,28], 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 14 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 [22], 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

In recent years, numerous research groups have addressed the issue of adaptive middleware frameworks that can accommodate dynamic, heterogeneous infrastructures. Examples include Adapt [29], MOST [1], Rover [30], MASH [31], TAO [2], dynamicTAO [32], MobiWare [33], MCF [3], QuO [4], MPA [34], Odyssey [35], DaCapo++ [5], RCSM [36], CEA [23], and MOOSCo [37]. In addition, several higher-level frameworks have been designed to support wearable or ubiquitous applications; examples include Hive [8], Ektara [9], and Proem [10], Puppeteer [12], Aura [11], and the Context Toolkit [13].

These projects and others have greatly improved the understanding of how a system can adapt to changes in the environment and in user behavior and interactions. Our work in the RAPIDware project complements such contributions by focusing on principled approaches to adaptive software design that include programming language support and rigorous software engineering methods. Such support holds the promise that compile-time and run-time checks can be performed on the adaptive code in order to help ensure consistency and preservation of certain key properties as the system changes. Moreover, these techniques facilitate the run-time adaptation of the system in ways not anticipated during the original development.

Several adaptive middleware projects involve adaptive extensions to CORBA [2, 4, 29, 32, 36, 38]. For example, in the Adapt project at Lancaster [29], CORBA is extended to support open bindings, which enable manipulation and reconfiguration of communication paths through the use of object graphs. This mechanism could be used directly to implement dynamically composable services for FEC and other QoS-related functions. 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 [39, 40], standardized in CORBA 2.6, and used also in dynamicTAO [41]. 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, Andersson and Ritzau [42] describe a method for supporting dynamic update of Java programs, but that technique requires a modified JVM. Our "weaving" of adaptive code with the base appli-

cation is reminiscent of aspect-oriented programming [21]. Although many projects in the AOP community focus on compile-time weaving [43], a growing number of projects focus on run-time composition [44, 45]. 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 [46], 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 [47] 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 [14, 20], 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] A. Friday, N. Davies, G. Blair, and K. Cheverst, "Developing adaptive applications: The MOST experience," *Journal of Integrated Computer-Aided Engineering*, vol. 6, no. 2, pp. 143–157, 1999.

[2] 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.

[3] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE Journal of Selected Areas in Communications*, vol. 17, September 1999.

[4] 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.

[5] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A flexible middleware for multimedia communication: Design implementation, and experience," *IEEE Journal of Selected Areas in Communications*, vol. 17, pp. 1580–1598, September 1999.

[6] 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.

[7] S. Fickas, G. Kortuem, and Z. Segall, "Software organization for dynamic and adaptable wearable systems," in *Proceedings First International Symposium on Wearable Computers (ISWC'97)*, (Cambridge, Massachusetts), October 1997.

[8] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes, "Hive: Distributed agents for networking things," in *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[9] R. W. DeVaul and A. Pentland, "The Ektara architecture: The right framework for context-aware wearable and ubiquitous computing applications." The Media Laboratory, Massachusetts Institute of Technology, unpublished, 2000.

[10] G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. Segall, "When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad-hoc networks," in *Proceedings of the 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, (Linköpings, Sweden), August 2001.

[11] J. P. Sousa and D. Garlan, "Aura: an architectural framework for user mobility in ubiquitous computing environments," in *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, (Montreal, Canada), August 2000. to appear.

[12] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel, "Reducing the energy usage of office applications," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, (Heidelberg, Germany), pp. 252–272, November 2001.

[13] A. K. Dey and G. D. Abowd, "The Context Toolkit: Aiding the development of context-aware applications," in *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing*, (Limerick, Ireland), June 2000.

[14] 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. to appear.

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

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

[17] G. Kiczales, "Towards a new model of abstraction in the engineering of software," in *International Workshop on Reflection and Meta-Level Architecture*, (Tama-City, Tokyo, Japan), nov 1992.

[18] 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.

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

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

[21] 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.

[22] P. K. McKinley, U. I. Padmanabhan, and N. Ancha, "Experiments in composing proxy audio services for mobile users," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, (Heidelberg, Germany), pp. 99–120, November 2001.

[23] 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.

[24] P. K. McKinley and A. P. Mani, "An experimental study of adaptive forward error correction for wireless collaborative computing," in *Proceedings of the IEEE 2001 Symposium on Applications and the Internet (SAINT-01)*, (San Diego-Mission Valley, California), January 2001.

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

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

[27] 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.

[28] P. K. McKinley and S. Gaurav, "Experimental evaluation of forward error correction on multicast audio streams in wireless LANs," in *Proceedings of ACM Multimedia 2000*, (Los Angeles, California), pp. 416–418, November 2000.

[29] 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.

[30] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, "Mobile computing with the Rover toolkit," *IEEE Transactions on Computers: Special issue on Mobile Computing*, vol. 46, March 1997.

[31] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T. Tung, D. Wu, and B. Smith, "Toward a common infrastructure for multimedia-networking middleware," in *Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97), St. Louis, Missouri*, May 1997.

[32] 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.

[33] O. Angin, A. T. Campbell, M. E. Kounavis, and R.R.-F.M. Liao, "The Mobiware toolkit: Programmable support for adaptive mobile networking," *IEEE Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, August 1998.

[34] M. Roussopoulos, P. Maniatis, E. Swierk, K. Lai, G. Appenzeller, and M. Baker, "Person-level routing in the mobile people architecture," in *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, (Boulder, Colorado), October 1999.

[35] B. D. Noble and M. Satyanarayanan, "Experience with adaptive mobile applications in Odyssey," *Mobile Networks and Applications*, vol. 4, pp. 245–254, 1999.

[36] S. S. Yau and F. Karim, "Adaptive middleware for ubiquitous computing environments," in *Proceedings of IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES 2002)*, (Montreal, Canada), August 2002. to appear.

[37] H. Miranda, M. Antunes, L. Rodrigues, and A. R. Silva, "Group communication support for dependable multiuser object-oriented environments," in *SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, (Nürnberg, Germany), October 2000.

[38] C. Becker and K. Geihs, "Quality of service and o.o. oriented middleware multiple concerns and their separation," in *Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures*, (Phoenix, Arizona), April 2001.

[39] D. S. Nanbor Wang, Kirthika Parameswaran, "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.

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

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

[42] 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.

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

[44] 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.

[45] 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.

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

[47] 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.