SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:1157–1178
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.748

MetaSockets: design and operation of runtime reconfigurable communication services

S. M. Sadjadi[1,*,†], P. K. McKinley[2], E. P. Kasten[2] and Z. Zhou[2]

[1]*Autonomic Computing Research Laboratory, School of Computing and Information Sciences, Florida International University, Miami, FL 33199, U.S.A.*
[2]*Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, U.S.A.*

SUMMARY

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 runtime 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. Copyright © 2006 John Wiley & Sons, Ltd.

Received 19 April 2004; Revised 22 March 2006; Accepted 23 March 2006

KEY WORDS: mobile computing; quality-of-service; computational reflection; adaptive middleware; real-time audio-streaming; event-based systems

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

*Correspondence to: S. M. Sadjadi, Autonomic Computing Research Laboratory, School of Computing and Information Sciences, Florida International University, Miami, FL 33199, U.S.A.
†E-mail: sadjadi@cis.fiu.edu

Contract/grant sponsor: U.S. Department of the Navy and Office of Naval Research; contract/grant number: N00014-01-1-0744
Contract/grant sponsor: National Science Foundation; contract/grant numbers: CCR-9912407, EIA-0000433, EIA-0130724 and ITR-0313142

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 services to address changing conditions at runtime. Developing and maintaining such software is a non-trivial task. In this paper, we demonstrate the effectiveness of programming language support for the development and maintenance of an underlying communication infrastructure that must adapt to its environment.

Adaptability can be implemented in different parts of the system [1] (for more information, please refer to [2]). One approach is to introduce a layer of adaptive middleware between applications and underlying transport services [3–8]. 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 [9], an extension to Java that supports dynamic reconfiguration of software components. This paper concerns an Adaptive Java component called the *MetaSocket* (for 'metamorphic' socket). Although the socket abstraction is relatively low-level compared with many current distributed computing platforms (e.g. CORBA, Java RMI, DCOM, and .NET Remoting), its ubiquity in distributed applications, as well as in middleware platforms, makes it a natural place to introduce adaptive behavior. MetaSockets are created from existing Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli. For example, MetaSockets can dynamically enhance error control when the channel quality is poor and can use stronger encryption in hostile environments.

A key concept in the MetaSocket model is that adaptive functionality related to communication streams, possibly tangled throughout application code, is extracted and placed in the MetaSocket layer. Application modules and higher-level middleware layers can invoke traditional socket operations using MetaSockets, while the MetaSockets themselves can adapt (or be adapted) to changes in the environment. This *separation of concerns* [10] depicted in Figure 1, leads to code that is easier to maintain and evolves to incorporate new adaptive functionality.

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, 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 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.

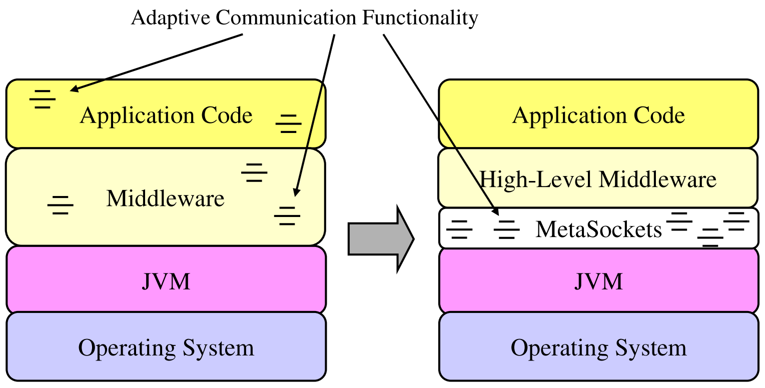

Figure 1. Separation of concerns using MetaSockets.

2. ADAPTIVE JAVA BACKGROUND

Adaptive Java [9] 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 [11,12], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. A key issue that arises in the application of reflection to middleware platforms is the degree to which the system should be able to change its own behavior. A completely open implementation implies that an application can be recomposed entirely at runtime. In the extreme, all the default components of the system can be destroyed and new ones instantiated, such that the goal of the base-level computation is changed (a spreadsheet can be recomposed as a video player!). On the other hand, limiting adaptability also limits the ability of the system to survive adverse situations.

We began our investigation of this problem by focusing on the reflective interfaces exhibited by components. In object-oriented environments, the entities at 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 (MOP). Rather than considering MOPs as orthogonal portals into base-level functionality, we consider a model in which MOPs are constructed from a set of primitive operations, or *atoms*, that provide access to component behavior. As shown in Figure 2, while different MOPs are defined for different aspects of adaptive behavior (e.g., fault tolerance, security, quality-of-service, power consumption), they are likely to overlap in their use of these atoms. This design appears to exhibit several desirable features. First, explicitly defining intersections in MOP functionality may facilitate coordinated adaptation to events. Second, additional MOPs can be constructed to address issues that did not arise in the original design. Third, limiting interaction with the base level may improve the ability of the system to check, at run-time, the consistency of modifications with the specified behavior of the component.

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

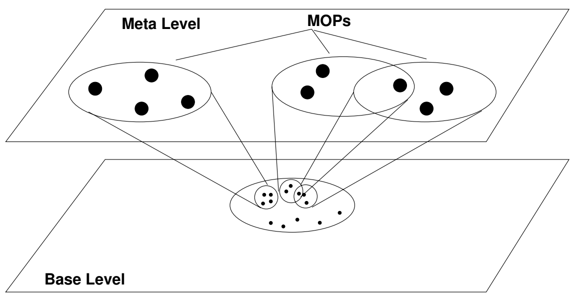

Figure 2. MOPs implemented with primitive operations.

behavior (*intercession*). Operations in the computation dimension are referred to as *invocations*. Operations in the introspection dimension are called *refractions*; they offer a partial view of internal structure and behavior, but are not allowed to change the state or behavior of the component. Operations in the intercession dimension are called *transmutations*; they are used to modify the computational behavior of the component.

An existing Java class can be converted into an adaptable component in two steps. In the first step, a *base-level* Adaptive Java component is constructed from the Java class through an operation called *absorption*, which uses the absorbs keyword. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added to and removed from existing components at run time using meta-level transmutations. In the second step, *metafication* enables the creation of refractions and transmutations that operate on the base component. Meta components are defined using the metafy keyword. The meta-level can also be given a meta-level (meta-meta-level), which can be used to refract and transmute the meta-level. In theory, this reification of meta-levels for other meta-levels could continue indefinitely [12]. Example code is provided in Section 3.2.

Adaptive Java Version 1.0 [9] is implemented using CUP [13], 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 MetaSocket. Our discussion is limited to particular types of MetaSocket designed to enhance the quality of service for multicast communication streams. However, the MetaSocket model is general: MetaSockets can also be used for unicast communication and can be tailored to provide adaptive functionality in other cross-cutting concerns, such as security, energy consumption, and fault tolerance.

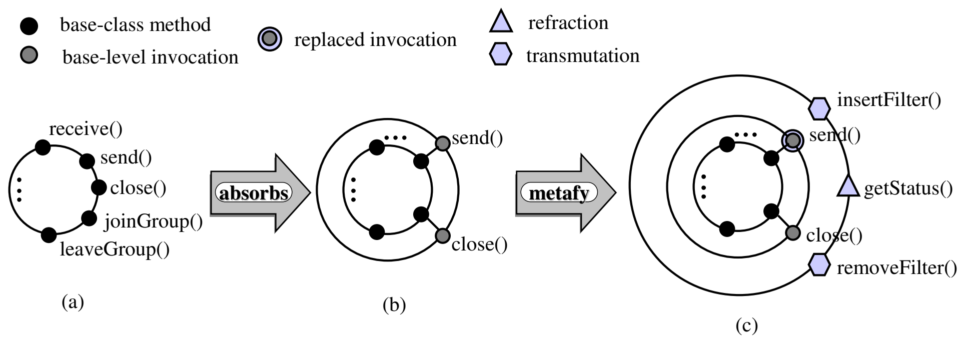

Figure 3. MetaSocket absorption and metafication: (a) Java MulticastSocket as the base-level class; (b) SendMSocket as the base-level component; (c) MetaSendMSocket, a filter-oriented meta-level component.

Figure 3 shows the absorption of a Java MulticastSocket base-level class by a SendMSocket base-level component, and the metafication of this component to a MetaSendMSocket meta-level component. Figure 3(a) depicts a Java MulticastSocket class and a subset of its public methods: receive(), send(), close(), joinGroup(), and leaveGroup(). Figure 3(b) shows a SendMSocket component, which is designed to be used as a *send-only* multicast socket. The SendMSocket component *absorbs* the Java MulticastSocket class and implements send() and close() invocations that can be used by other components. Other methods of the base-level class are occluded. A link between an invocation and a method indicates a dependency. For example, the send() invocation depends on the send() method, because its implementation calls that method. Figure 3(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 enables Adaptive Java applications to adapt to unanticipated changes at run time.

3.1. Internal architecture and operation

Figure 4 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,

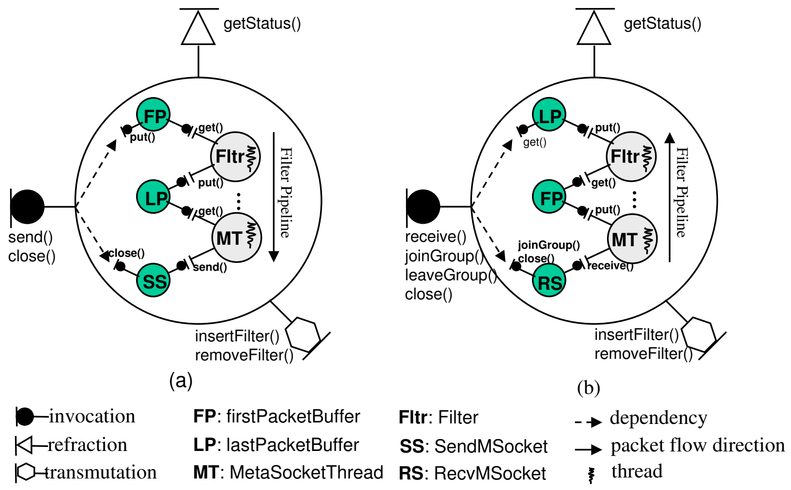

Figure 4. MetaSocket internal architecture: (a) MetaSendMSocket, a send-only metamorphic multicast socket; (b) MetaRecvMSocket, a receive-only metamorphic multicast socket.

each of which processes the packet. To keep the architecture simple, this pipeline is the same for all packets. The pipeline has two modes of operation, namely, MultiThreaded and SingleThreaded. In the former mode, each filter runs as a separate thread so that filters can execute concurrently to process different packets; while in the latter mode, a single thread executes all the filters and processes packets one-by-one through the pipeline.

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 at the sender side that needs a reverse operation at the receiver side, an application-level header is prepended to the packet after it is processed. At the receiver side, these headers identify the reverse processing order and, as a result, filters required to reverse the transformations will be added to the pipeline. Each header contains enough information to find and load the receiving filters using Component Loader and Trader components explained later in Section 4.

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. As a packet buffer may be used by multiple threads, its invocations, including get() and put(), are defined as synchronized.

Each filter in the filter pipeline retrieves a packet from its source packet buffer, processes it, and places it into its destination packet buffer. The destination packet buffer of a filter in the pipeline is either the source packet buffer of the next filter or lastPacketBuffer.

Inserting and removing filters

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

Sender operation

Let us consider the sender, as shown in Figure 4(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 4(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

```
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 5. Excerpted code for SendMSocket.

replaces the base-level receive() invocation with the meta-level receive() invocation defined by MetaRecvMSocket. Instead of calling the RecvMSocket receive() invocation, the MetaRecvMSocket receive() invocation retrieves packets directly from lastPacketBuffer. Before returning the packet to the caller, however, the receive() invocation checks the packet's MetaSocket header. If a terminator header is found at the beginning of the packet, then receive() removes this header and returns the original packet to the caller. Otherwise, additional filter processing needs to be performed on the packet before delivering it to the application. In this case, receive() generates a FilterMismatchEvent event containing the packet and the position of the required Filter in the filter pipeline. (Every filter at the receiving side performs a similar task and compares the filter ID of the next packet to its ID.) This event is sent to the *EventMediator*, a singleton component in each address's space that decouples event generators from event listeners [14]. 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. In this manner, filters developed by third parties are not required to handle any packet headers other than their own. This logic is encapsulated in the other components of the MetaSockets. Additional details on event handling are discussed in the next section.

3.2. Syntax of absorption and metafication

Figure 5 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.

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 6. At any point during the execution of the application, a running SendMSocket component can be metafied by calling its constructor.

```
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 6. Excerpted code for MetaSendMSocket.

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.

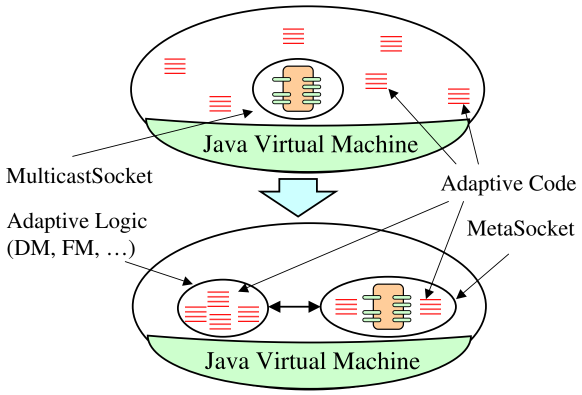

Figure 7. Example of separation of concerns using MetaSockets.

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* [10], depicted in Figure 7, 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.

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 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 8.

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 ms of audio data; relatively small packets are necessary to reduce jitter and minimize losses.

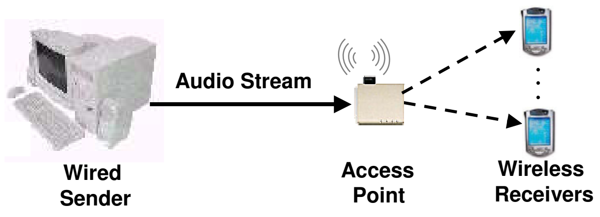

Figure 8. Physical experimental configuration.

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 illustrates the major parts of the receiving side of the ASA; the sending side has a similar structure. Note that we introduce new notations to distinguish the type of interaction among components (one for invocations and another for refractions and/or transmutations). Most of the receiving system executes on an iPAQ handheld computer, but one program, called a *Trader*, executes on a desktop workstation. The two systems communicate over the WLAN. In Adaptive Java, every 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 9 comprises five main components: a Player, a DecisionMaker, an EventMediator, a ComponentLoader, and a MetaRecvMSocket. Except for the Player component, which is specific to ASA, the other components are part of the AdaptiveJava component infrastructure that can be reused in any other application. 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 non-functional behavior of the subcomponents of its container component. DMs are arranged hierarchically, such that a given DM inherits rules from a higher-level DM and might provide rules to lower-level DMs. (In our simple example application, the main component on the iPAQ contains a single DM.) Depending on its rules and the current 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 dynamically to load components is especially important for mobile devices, where resources might be limited and overheads should be minimized.

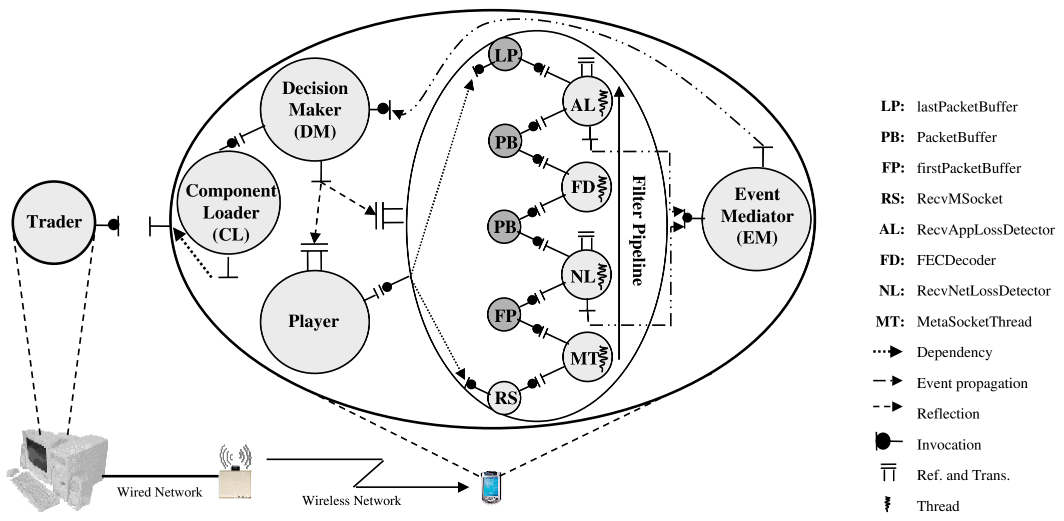

Figure 9. Interaction among components in the audio streaming application.

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* [14]. 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

A number of commonly used filters have been developed and added to the Adaptive Java filter library. In this case study, we used only two types of filter 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 used widely in wireless networks, where factors such as signal strength, interference, and antennae alignment produce dynamic and location-dependent packet losses. In current wireless LANs,

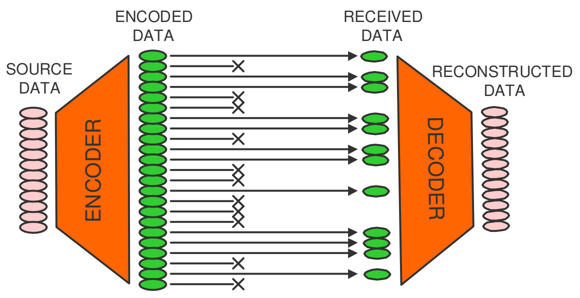

Figure 10. Operation of block erasure code.

these problems affect multicast connections more than unicast connections, as the 802.11b MAC layer does not provide link-level acknowledgments 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 [15]. As shown in Figure 10, 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 [15]. These codes are ideal for wireless multicasting, as 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 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 11, metafy the FECEncoder and FECDecoder components, respectively. Each provides a getNK() refraction and setNK() transmutation, which are used at run time to read and set the values of n and k. If a packet arrives with a different n or k value than is expected, the MetaFECDecoder fires a FECMismatchNKEvent event. In response, the DM uses setNK() transmutation and adjusts the values for k and n appropriately.

The second type of filter used in our case study monitors events related to packet loss 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 owing to the use of FEC. The metafied versions of these filters is shown in Figure 12. 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

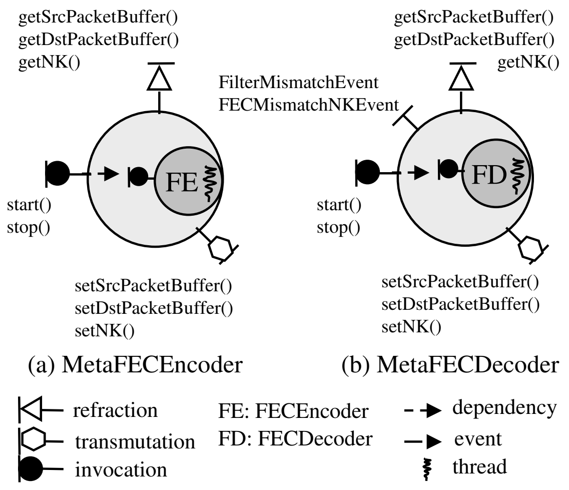

Figure 11. Design of forward error correction filters.

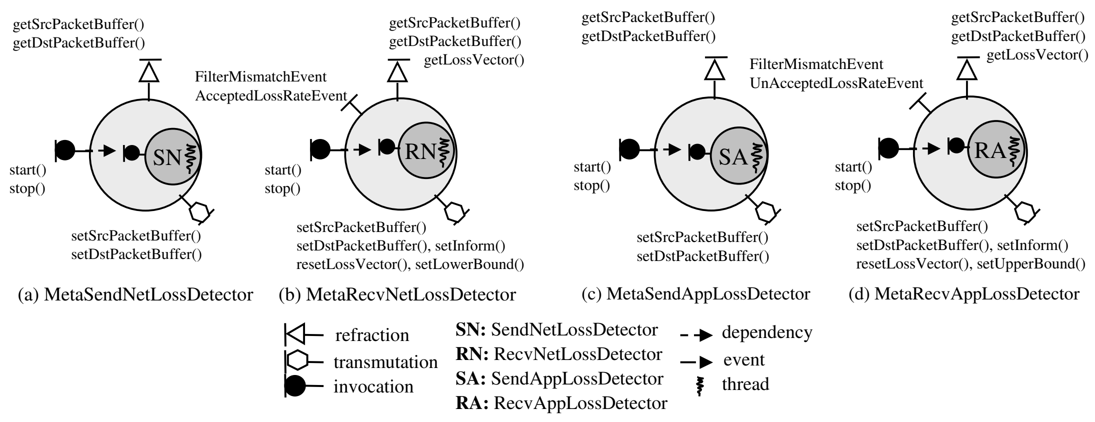

Figure 12. Design of packet loss monitoring filters.

the identifier of the corresponding peer filter on the receiver. Each filter on the receiver uses sequence numbers to calculate the packet loss rate over a specified window in the packet stream and stores this information in a vector. Metafying these components provides refractions and transmutations to read the current loss rate and to set or change upper and lower thresholds with respect to the loss rate.

The sender's DM (the global DM) and the receiver's DM (the local DM) work together and use a simple set of rules to make decisions about the use of filters and changes in their behavior.

If the loss rate observed by the application rises above a specified threshold, then the global DM can decide to insert an FEC filter in the pipeline or modify the (n, k) 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.

In the next section, we demonstrate how insertion and removal of FEC filters adapt the behavior of MetaSocket to respond dynamically to the changes in the wireless network. For experimental results on the effect of changes in the n and k parameters on the quality of service and energy consumption of mobile devices, please refer to [16].

5. PERFORMANCE EVALUATION

To evaluate the effect of MetaSockets on the performance of audio streaming, we conducted several experiments using ASA. First, we report the effect of using MetaSockets in an environment with simulated packet loss, followed by results with real packet loss on a mobile computing testbed. We note that, while MetaSocket reconfiguration incurs processing overhead, the results reported in this section demonstrate that the overhead is not significant enough to affect the real-time behavior of the audio-streaming application.

5.1. Adapting to simulated packet loss

One well-known difficulty in conducting experimental research in wireless environments is the ability to reproduce results, given the highly dynamic nature of the medium [17]. In this set of tests, we created artificial losses by dropping packets in software according to a predefined loss function. In this way, we are able to compare the effects of different parameter settings on the behavior of MetaSockets.

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

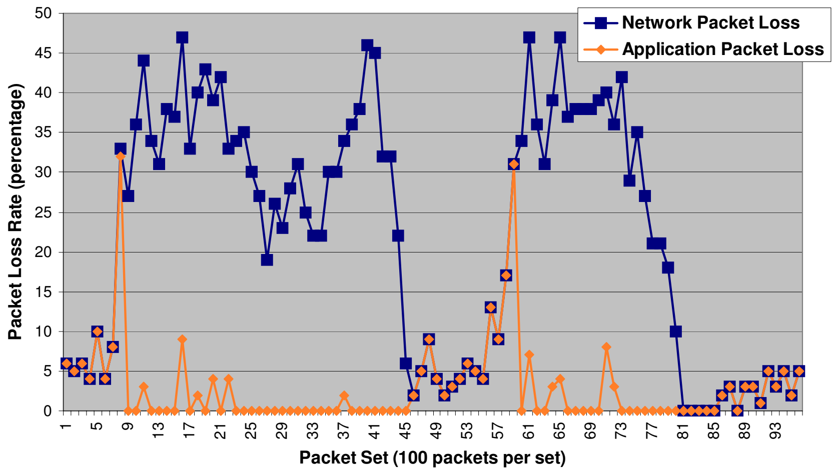

Figure 13. MetaSockets performance in an environment with real packet loss for the FEC (8, 4) code.

Figure 13 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.

As shown in Figure 13, the FEC (8, 4) code is very effective in reducing the packet loss rate as observed by the application from packet set 8 to packet set 45. At packet set 45, the RecvNetLossDetector detects that the loss rate has dipped below the 10% lower threshold, so it fires an AcceptableLossRateEvent. In response, the local DM sends a request to the global DM to remove the FEC filter. The DM complies, as under low-loss conditions, the 100% overhead of an FEC (8, 4) 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.

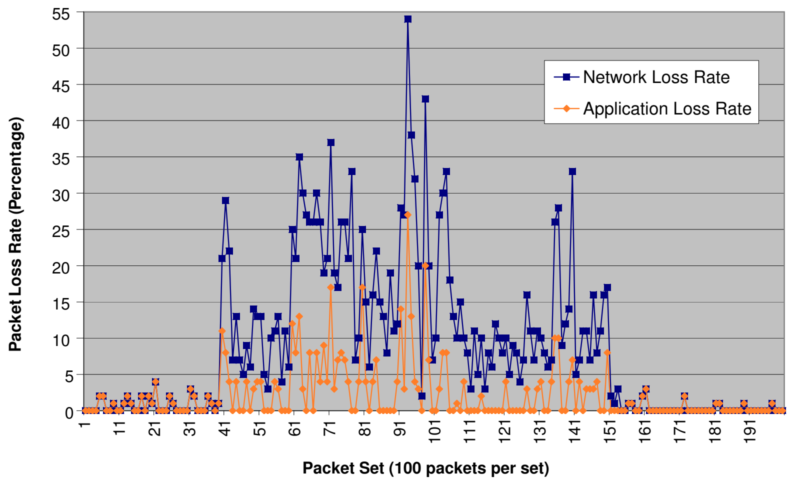

Figure 14. MetaSockets performance in an environment with real packet loss for the FEC (20, 4) code. This experiment was conducted while a mobile user was moving about the hallway adjacent to the wireless access point.

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 13 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.

5.2. Adapting to real packet loss

Figure 14 provides a trace of an experiment, with real packet losses, that demonstrates how MetaSockets adapt to loss rates due to user motion. One user sits at a desktop workstation in our research lab and speaks, while another listens on an iPAQ as he moves about an adjacent hallway. The loss rate is very high while the user is moving. In this particular test, the iPAQ user stood outside the lab for approximately 30 s, walked up and down the hall for another 90 s, then stood relatively still for another 30 s. The upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 10%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 1%. Figure 14 plots the packet loss as observed by the two loss-monitoring filters on the receiver iPAQ. When the program begins execution, the sending process inserts a SendAppLossDetector filter into its MetaSocket, which quickly causes the receiver to insert the corresponding RecvAppLossDetector. As shown in Figure 14, the loss rate is low at the beginning of the test, then increases quickly when the user starts walking.

The RecvAppLossDetector filter detects that the loss rate has passed the upper threshold of 10% and fires an UnAcceptableLossRateEvent. The DM decides, based on its set of rules, to insert two filters, an FECEncoder filter with default parameters ($n = 20$ and $k = 4$ in this particular test), and a SendNetLossDetector filter. When packets containing the headers of the two new filters begin arriving at the receiver, the RecvAppLossDetector detects a packet header that does not match its own identifier. It fires a FilterMismatchEvent at two different times, one for each new packet type. These events result in the insertion of a RecvNetLossDetector filter and a FECDecoder filter in the opposite order as at the sender.

As shown in Figure 14, the FEC (20, 4) code is effective in reducing the packet loss rate as observed by the application. The average loss rate in the absence of FEC filters is about 16%, while in the presence of FEC filters the loss rate is improved to 3.5%. Near packet 15 200 the RecvNetLossDetector detects that the loss rate has dipped below the 1% lower threshold, so it fires an AcceptableLossRateEvent. In response, the local DM sends a request to the global DM to remove the FEC filter. The DM complies, since under low-loss conditions, the high overhead of an FEC (20, 4) code simply wastes bandwidth and energy. It also removes the SendNetLossDetector filter in order to minimize data stream processing under favorable conditions. The arrival of packets without the two headers produces two FilterMismatchEvent events at the receiving side, and the peer filters are removed. As a result, the loss rate experienced by the application is again the same as the network loss rate for the remainder of the experiment. In our ongoing studies, we are investigating such adaptive behavior and the role of other concerns, such as security policies and remaining battery lifetime, which affect the relative importance of QoS.

Figure 15 shows a trace of an experiment that demonstrates how MetaSockets adapt to loss rates that depend on receiver location, relative to the source. On average, the loss rate without adaptation is about 26%, while with adaptation it is about 8%. A stationary user speaks into a laptop microphone, while another user listens on an iPAQ as he moves among locations in the wireless cell. In this particular test, the iPAQ user remains in a low packet loss area for approximately 30 min, moves to a high packet loss area for another 40 min, moves back to the low packet loss location for another 30 min, then re-enters the high packet loss location. He remains there until the iPAQ's external battery drains and the WNIC is disconnected. As shown in Figure 15, the FEC (4, 2) code is effective in reducing the packet loss rate as observed by the application. In this experiment, the upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 20%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 5%. Moreover, the upper threshold for the LifeTimeEstimate to generate an AcceptableLifeTimeEvent is 200 min, and the lower threshold to generate an UnAcceptableLifeTimeEvent is 170 min. When the user first enters the high loss area, at time 30, the RecvAppLossDetector filter detects that the loss rate has passed the upper threshold of 20% and fires an UnAcceptableLossRateEvent. The DM decides, based on its set of rules, to insert an FECEncoder filter, resulting in reducing the packet loss rate. When the user moves back to the low loss area at time 70, the DM decides to remove the FECEncoder, as the overhead of FEC simply wastes bandwidth and energy. At time 137, the LifeTimeEstimate determines that the remaining battery capacity is not sufficient for maintaining the communication between users while providing QoS support through FEC. In response, an UnAcceptableLifeTimeEvent is fired and the DM decides to remove the FECEncoder. We also measured the remaining battery capacity during the above experiment and for a non-adaptive trace. The adaptive version extends the battery lifetime by approximately 27 min.

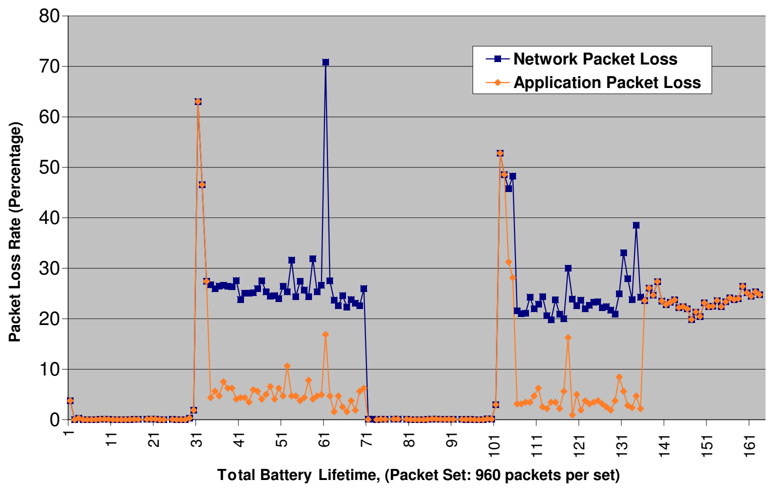

Figure 15. MetaSockets performance in an environment with real packet loss for the FEC (4, 2) code. This experiment shows the adaptability of MetaSockets not only to the packet loss rate, but also to the remaining battery lifetime.

6. RELATED WORK

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

The first category includes middleware projects that support adaptive behavior in Java programs by extending the Java Virtual Machine. Examples include Iguana/J [18], Meta Java [19], JDrums [20], Guaraná on Java [21], PROSE [22], and R-Java [23]. A major benefit of implementing adaptation in this way is that the execution of virtually any bytecode instruction can be intercepted within a customized JVM. In contrast, only messages originally targeted for Java sockets can be intercepted and adapted dynamically using MetaSockets. However, some researchers have noted that fine-grained interception at the JVM level can produce significant performance overhead. For example, according to [18], the time for common operations such as creating new objects can be increased by an order of magnitude. Another advantage of JVM-supported adaptation is that it is usually transparent to the target Java program (no code modification required). On the other hand, using a custom JVM tends to limit portability. Since our implementation of Adaptive Java uses source-to-source compilation, MetaSockets can execute atop any standard JVM. Moreover, to address the transparency issue, we have recently developed a generator framework, called TRAP/J [24], which enables adaptable components such as MetaSockets to be woven into existing Java programs without modifying the application source code.

The second category includes projects that use aspect-oriented programming [10] to weave adaptive code into functional code. Although many projects in the AOP community address compile-time weaving [25], a growing number of projects focus on run-time composition [26–31]. 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 [32], which provide a mechanism for disentangling the cross-cutting concerns of a software system. This system declares filters that intercept messages received and sent by objects. As such, messages can be massaged and checked before they are delivered to an object, separating aspects, such as security authentication or bounds checking, from the objects that send and receive these messages. Adaptive Java's approach to composition using encapsulation could be used to instantiate a message-filtering design where components are extended and invocations added such that a call to an invocation would be filtered through subsequent encapsulation layers. However, such a design would not have the source code expressiveness provided by the declarative specification language in composition filters.

The third category of related work includes projects that, like Adaptive Java, extend the Java syntax and provide new constructs to allow developers to write adaptable applications more expressively. Examples include PCL [33], Open Java [34], R-Java [23], and Handi-Wrap [35]. Open Java provides an approach supporting customized compilers that define new compile-time MOPs [36]. For example, to support writing expressive programs that use a set of design patterns, Open Java enables a developer to build a customized compiler that understands the new syntax. The PCL project [33] also focuses on language support for run-time adaptability. Our concept of 'wrapping' classes with base components is similar to the use of *Adaptors* used in PCL. However, modification of the base class in PCL appears to be limited to changing variable values, whereas Adaptive Java transmutations can modify arbitrary structures or subcomponents. Moreover, by combining encapsulation with metafication, Adaptive Java can be used to realize adaptations in multiple meta-levels.

7. CONCLUSIONS AND FUTURE DIRECTIONS

In this work, we investigated the use 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. 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.

As part of our ongoing research, we have developed a software tool, called TRAP/J [37], that enables new adaptable behavior to be added to existing Java applications transparently (that is, without modifying the application source code and without modifying the JVM). The generation process combines behavioral reflection and aspect-oriented programming to achieve this goal. Specifically, TRAP/J enables the developer to select, at compile time, a subset of classes in the existing program that are to be adaptable at run time. TRAP/J then generates specific aspects and reflective classes associated with the selected classes, producing an *adapt-ready* program. As the program executes, new behavior can be introduced via interfaces to the reflective classes.

While this paper demonstrated the application of MetaSockets to a specific communication service, we emphasize that these 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 and TRAP/J 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 can be found at http://www.cse.msu.edu/sens and http://www.cs.fiu.edu/acrl. Papers and other results related to the RAPIDware project, including a download of the MetaSockets and Adaptive Java source code, are available at http://www.cse.msu.edu/rapidware.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of the Navy and Office of Naval Research (ONR) grant number N00014-01-1-0744 and in part by National Science Foundation (NSF) grant numbers CCR-9912407, EIA-0000433, EIA-0130724, and ITR-0313142. This work was done while SMS was with Michigan State University.

REFERENCES

1. McKinley PK, Sadjadi SM, Kasten EP, Cheng BHC. Composing adaptive software. *IEEE Computer* 2004; **37**(7):56–64.
2. McKinley PK, Sadjadi SM, Kasten EP, Cheng BHC. A taxonomy of compositional adaptation. *Technical Report MSU-CSE-04-17*, Department of Computer Science, Michigan State University, East Lansing, MI, May 2004. Available at: http://www.cs.fiu.edu/∼sadjadi/Publications/CompositionalAdaptationTaxonomy-TechRep.pdf.
3. Schmidt DC, Levine DL, Mungee S. The design of the TAO real-time object request broker. *Computer Communications* 1998; **21**:294–324.
4. Kon F, Román M, Liu P, Mao J, Yamane T, Magalhães LC, Campbell RH. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*. ACM Press: New York, 2000.
5. Zinky JA, Bakken DE, Schantz RE. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems* 1997; **3**(1):55–73.
6. Blair GS, Coulson G, Robin P, Papathomas M. An architecture for next generation middleware. *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998. Springer: Berlin, 1998.
7. Ledoux T. *OpenCorba: A Reflective Open Broker* (*Lecture Notes in Computer Science*, vol. 1616). Springer: Berlin, 1999.
8. Baldoni R, Marchetti C, Termini A. Active software replication through a three-tier approach. *Proceedings of the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, October 2002. IEEE Computer Society Press: Washington, DC, 2002; 109–118.
9. Kasten E, McKinley PK, Sadjadi S, Stirewalt R. Separating introspection and intercession in metamorphic distributed systems. *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, July 2002. IEEE Computer Society Press: Washington, DC, 2002; 465–472.
10. Kiczales G, Lamping J, Mendhekar A, Maeda C, Videira Lopes C, Loingtier JM, Irwin J. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (*Lecture Notes in Computer Science*, vol. 1241). Springer: Berlin, 1997.
11. Smith BC. Reflection and semantics in Lisp. *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*. ACM Press: New York, 1984; 23–35.
12. Maes P. Concepts and experiments in computational reflection. *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*. ACM Press: New York, 1987.

13. Hudson SE (ed.). *CUP User's Manual*. Usability Center, Georgia Institute of Technology: Atlanta, GA, 1999.
14. Bacon J, Moody K, Bates J, Hayton R, Ma C, McNeil A, Seidel O, Spiteri M. Generic support for distributed applications. *IEEE Computer* 2000; **33**(3):68–76.
15. Rizzo L. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review* 1997; **27**:24–36.
16. Zhou Z, McKinley PK, Sadjadi SM. On quality-of-service and energy consumption tradeoffs in fec-enabled audio streaming. *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS 2004)*, June 2004. IEEE Communications Society: New York, 2004. (Winner of the IWQoS 2004 best student paper award.)
17. Eckhardt DA, Steenkiste P. A trace-based evaluation of adaptive error correction for a wireless local area network. *Mobile Networks and Applications* 1999; **4**(4):273–287.
18. Redmond B, Cahill V. Supporting unanticipated dynamic adaptation of application behaviour. *Proceedings of the 16th European Conference on Object-Oriented Programming*, June 2002. Springer: London, 2002; 205–230.
19. Golm M. Design and implementation of a meta architecture for Java. *Master's Thesis*, Friedrich-Alexander-University, Erlangen-Nurenburg, January 1997.
20. Andersson J, Ritzau T. Dynamic code update in JDrums. *Proceedings of the ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing*. ACM Press: New York, 2000.
21. Oliva A, Buzato LE. The implementation of Guaraná on Java. *Technical Report IC-98-32*, IC-Unicamp, September 1998.
22. Popovici A, Gross T, Alonso G. Dynamic homogenous AOP with PROSE. *Technical Report*, Department of Computer Science, Federal Institute of Technology, Zurich, 2001.
23. de Oliveira Guimarães J. Reflection for statically typed languages. *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Springer: London, 1998; 440–461.
24. Sadjadi SM, McKinley PK, Cheng BH, Stirewalt RK. TRAP/J: Transparent generation of adaptable Java programs. *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, October 2004. Springer: Heidelberg, 2004.
25. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. *An Overview of AspectJ* (*Lecture Notes in Computer Science*, vol. 2072). Springer: Berlin, 2001; 327–355.
26. Truyen E, Jörgensen BN, Joosen W, Verbaeten P. Aspects for run-time component integration. *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia Antipolis and Cannes, France. Springer: Berlin, 2000.
27. Akkai F, Bader A, Elrad T. Dynamic weaving for building reconfigurable software systems. *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, October 2001. ACM Press: New York, 2001.
28. Ren S, Beckman M, Elrad T. System imposed and application compliant adaptation. *Proceedings of the 4th IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, March 2004. IEEE Computer Society Press: Washington, DC, 2004.
29. Wagelaar D. Towards a context-driven development framework for ambient intelligence. *Proceedings of the 4th IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, March 2004. IEEE Computer Society Press: Washington, DC, 2004.
30. Hirschfeld R, Kawamura K. Dynamic service adaptation. *Proceedings of the 4th IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04)*, March 2004. IEEE Computer Society Press: Washington, DC, 2004.
31. Welch I, Stroud RJ. Kava—a reflective Java based on bytecode rewriting. *Reflection and Software Engineering* (*Lecture Notes in Computer Science*, vol. 1826), Cazzola W, Stroud RJ, Tisato F (eds.). Springer: Heidelberg, 2000; 157–169.
32. Bergmans L, Aksit M. Composing crosscutting concerns using composition filters. *Communications of the ACM* 2001; **44**(10):51–57.
33. Adve V, Lam VV, Ensink B. Language and compiler support for adaptive distributed applications. *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*. ACM Press: New York, 2001.
34. Tatsubori M, Chiba S, Itano K, Killijian MO. OpenJava: A class-based macro system for Java. *Proceedings of Reflection and Software Engineering* (*Lecture Notes in Computer Science*, vol. 1826). Springer: Berlin, 1999; 117–133.
35. Baker J, Hsieh W. Runtime aspect weaving through metaprogramming. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 2002. ACM Press: New York, 2002; 86–95.
36. Caromel D, Vayssière J. Reflections on MOPs, Components, and Java Security. *Proceedings of ECOOP 2001* (*Lecture Notes in Computer Science*, vol. 2072), Knudsen JL (ed.). Springer: Berlin, 2001; 256–274.
37. Sadjadi SM, McKinley PK, Stirewalt REK, Cheng BH. Generation of self-optimizing wireless network applications. *Proceedings of the International Conference on Autonomic Computing (ICAC-04)*, May 2004. IEEE Computer Society Press: Washington, DC, 2004; 310–311.