Composable Proxy Services to Support Collaboration on the Mobile Internet

Philip K. McKinley, *Member*, *IEEE*, Udiyan I. Padmanabhan, Nandagopal Ancha, and Seyed Masoud Sadjadi, *Student Member*, *IEEE*

Abstract—This paper describes the design and operation of a composable proxy infrastructure that enables mobile Internet users to collaborate via heterogeneous devices and network connections. The approach is based on detachable Java I/O streams, which enable proxy filters and transcoders to be dynamically inserted, removed, and reordered on a given data stream. Unlike conventional Java I/O streams, detachable streams can be stopped, disconnected, reconnected, and restarted. As such, they provide a convenient method by which to support the dynamic composition of proxy services. Moreover, use of the I/O stream abstraction enables network distribution and stream adaptability to be implemented transparently with respect to application components. The operation and implementation of detachable streams are described. To evaluate the composable proxy infrastructure, it is used to enhance interactive audio communication among users of a Web-based collaborative computing framework. Two forward error correction (FEC) proxylets are developed, one using block erasure codes and the other using the GSM 06.10 encoding algorithm. Separately, each type of FEC improves the ability of the audio stream to tolerate errors in a wireless LAN environment. When composed in a single proxy, however, they cooperate to correct additional types of burst errors. Results are presented from a performance study conducted on a mobile computing testbed.

Index Terms—Adaptive middleware, heterogeneous collaborative computing, mobile hosts, wireless local area networks, forward error correction, interactive audio streams, component-based design.

1 Introduction

Ubiquitous data access is becoming reality due to the large-scale deployment of wireless communication services and advances in mobile computing devices. While cellular telephony is the major contributor to this revolution, many applications demand higher bandwidth than cellular networks are likely to provide in the near future. As a result, wireless local area networks (WLANs) are available in many hotels, airports, schools, homes, and businesses. This "wireless edge" of the Internet includes those nodes that are one, or at most a few, wireless hops from the wired infrastructure. In addition to single-user applications, such as Web browsing and e-mail, this expanding mobile infrastructure will support a wide variety of *multiparty*, collaborative applications. Examples include computer-supported cooperative work, wireless instructional environments, and mobile operator support in large industrial and public facilities. A diverse communication environment enables individuals to collaborate via widely disparate technologies, some using workstations on high-speed local area networks (LANs), and others using wireless handheld or wearable devices.

The multicast-capable infrastructure offered by WLANs is well-suited to supporting collaborative applications, but, unfortunately, mobile computing environments exhibit operating conditions that differ greatly from their wired counterparts. In particular, the application must tolerate the highly dynamic channel conditions that arise as the users move about the environment. Moreover, the computing devices used by participants often vary in terms of display characteristics, processor speed, memory size, and battery lifetimes. Given their synchronous and interactive nature, collaborative applications are particularly sensitive to these differences. To enable effective collaboration among users in such environments, the communcation-related software must be able to adapt to these dynamic conditions at runtime.

One approach to accommodating heterogeneity and dynamic changes is to introduce a layer of *adaptive middleware* between applications and underlying transport services [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. The appropriate middleware framework can help to insulate application components from platform variations and changes in network conditions, minimize the state information that must be migrated during handoff from one device to another, and simplify the maintenance of security and fault tolerance invariants. Moreover, a properly designed middleware framework can facilitate the development of *new* applications through software reuse and domain-specific extensibility.

We are currently conducting a project called *RAPIDware* that addresses the design and implementation of middleware services for dynamic, heterogeneous environments. A major goal of the RAPIDware project is to develop adaptive mechanisms and programming abstractions that enable

- *P.K. McKinley and S.M. Sadjadi are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI. E-mail: {mckinley, sadjadis}@cse.msu.edu.*
- *U.I. Padmanabhan is with Microsoft Corporation, Redmond, WA. E-mail: diyanp@microsoft.com.*
- *N. Ancha is with Ericsson IP Infrastructure, Raleigh, NC. E-mail: Nandagopal.Ancha@ericsson.com*

Manuscript received 15 Oct. 2001; revised 28 Aug. 2002; accepted 30 Oct. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117660.

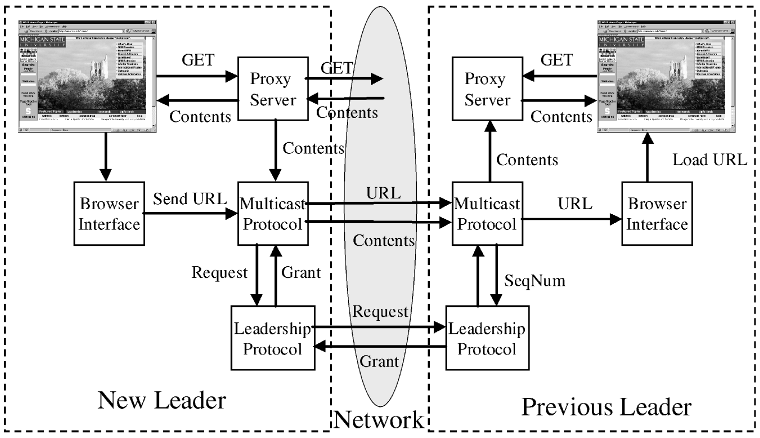


Fig. 1. Pavilion-based collaborative browsing.

middleware frameworks to execute in an autonomous manner, instantiating and reconfiguring components at runtime in response to the changing needs of client systems. Such functionality is particularly important in *proxy servers*, which are often used to mitigate the limitations of mobile hosts and their wireless connections [8], [11], [16], [17], [18], [19], [20]. Adopting the terminology of the IETF Task Force on Open Pluggable Edge Services (OPES) [21], proxies are composed of many *proxylets*, which are functional components that can be inserted and removed dynamically at runtime without disturbing the network state.

This paper describes the design and operation of a RAPIDware proxy infrastructure that uses a set of detachable Java I/O stream classes developed by our group. Unlike conventional Java I/O streams, these detachable streams can be stopped, disconnected, reconnected, and restarted. As such, they provide part of the "glue" needed to support the dynamic composition of proxylets. Moreover, use of the I/O stream abstraction provides a clean way to implement distribution and adaptability of intermediate processing components transparently with respect to application components. To evaluate the operation and performance of the composable proxy infrastructure, we apply it to the problem of enhancing interactive audio streams transmitted among users of a Web-based collaboration framework. Specifically, we show how the proxy infrastructure can be used to combine two independent forward error correction (FEC) proxylets, one using block erasure codes [22] and the other using the GSM 06.10 encoding algorithm for wireless telephones [23]. Separately, each type of FEC improves the ability of the audio stream to tolerate errors in a WLAN environment. However, by simply plugging both proxylets into the proxy, they cooperate in a synergistic manner to correct additional types of burst errors occurring on the wireless network.

The remainder of the paper is organized as follows: In Section 2, we discuss the Pavilion groupware framework used in this study to illustrate the effects of dynamically composable proxies. Section 3 describes the design and operation of detachable streams. Section 4 describes the individual operation of the two audio proxylets and their combined functionality when coupled in a single proxy server. Section 5 presents results of an experimental study on a mobile computing testbed. Section 6 discusses related work on composable proxy services. Section 7 presents our conclusions and discusses future directions for the RAPIDware project.

2 Background

To support our study of Web-based collaboration in heterogeneous wireless environments, we have developed an object-oriented groupware framework called Pavilion [24]. Pavilion can be used in a default mode in which it operates as a synchronous collaborative web browser [25]. As shown in Fig. 1, Pavilion uses four major components to provide a synchronous environment to its users: browser interfaces, a suite of multicast protocols, an extensible leadership protocol, and configurable proxy servers. By default, a group member acquires leadership by simply selecting a hyperlink in his/her browser. On the current leader's system, shown on the left in Fig. 1, the browser interface monitors the activities of the web browser. The interface is notified when a new URL is loaded by the browser, including the loading of special types of resources, such as PostScript files, PowerPoint presentations, and audio and video clips. The leader's browser interface reliably multicasts this URL to all participants, while the leader's proxy server multicasts the Web resource itself, as well as any embedded or linked files, to the proxy servers of the other group members. At each receiving system, the browser interface requests the local Web browser to load the new URL. The target Web browser will subsequently initiate retrieval of the file(s), via its proxy, which will return the requested item(s). Pavilion currently runs on desktop, laptop, wearable, and handheld computer systems. While users participate in a collaborative browsing session, they can speak with one another via interactive audio channels.

When multimedia applications such as Pavilion are executed in a wireless environment, proxies are often used to represent mobile hosts to the rest of the network [8], [11], [16], [17], [18], [19]. In the case of Pavilion, we have designed proxies to provide transcoding of data streams to reduce bandwidth [26], data caching for memory-limited handheld devices [27], and forward error correction for

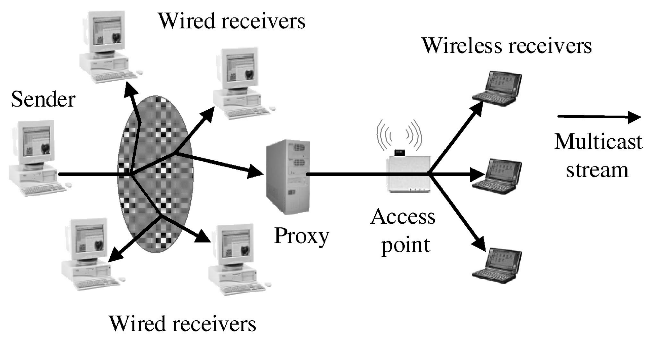


Fig. 2. Proxy configuration for nodes on a wireless LAN.

wireless video streaming [28], audio streaming [29], and reliable multicasting of Web resources [20]. However, those proxies were configured statically and did not adapt to changing conditions.

To illustrate the need for dynamic composition of proxies, let us consider the configuration shown in Fig. 2. Suppose that a proxy server is instantiated to support three mobile users, connected via a wireless LAN, who are collaborating via a Pavilion session with one another and with a set of users on the wired network. Among other duties, such a proxy might transcode images and video streams into lower-bandwidth formats prior to transmission on the wireless network. Now, let us assume that one of the users wants to maintain the connection, including a live audio stream from another user, as she moves from her office (near the wireless access point) to a conference room down the hall. The packet loss characteristics of WLANs can change dramatically over a distance of only several meters [20]. When losses rise above a given level, the proxy should insert an FEC proxylet into data streams in order to make them more resilient to losses. However, the insertion of the proxylet should not disturb the connections to the data sources and should take place only as needed in order to minimize bandwidth consumption.

The RAPIDware project addresses the dynamic reconfiguration of middleware components, including proxylets, in order to accommodate resource-limited hosts and changing network conditions. We focus on "lightweight" proxies, typically executed on workstations and other hosts accessible to the mobile user. While other projects have studied configurable proxies [8], [11], [17], [30], a key principle in RAPIDware is to separate adaptive middleware components from nonadaptive, or *core*, middleware services. We are investigating programming language abstractions that facilitate this separation and which enable third-party proxylets to be authenticated and dynamically inserted into an existing proxy. To manipulate data streams sent to and from clients' systems, we require mechanisms that enable the stream to be disconnected and redirected to another piece of code, without compromising the integrity of the data or that of the network state. In this paper, we explore the use of the Java I/O stream abstraction to achieve this goal, while providing transparency to the application and core middleware components. Next, we describe our proxy infrastructure, which is based on our detachable Java I/O streams. Following that, we describe a study in composing proxy services to improve the quality of interative audio streams among mobile users.

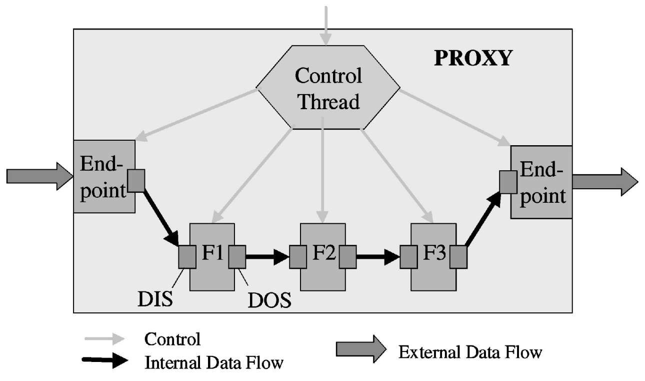


Fig. 3. Composition of proxy filters.

3 Proxy Infrastructure

We designed an infrastructure to enable *filters*, a particular type of proxylet described in Section 3.3, to be inserted, removed, and reordered [31]. Fig. 3 depicts an example RAPIDware proxy and its configuration for processing a single data stream. The proxy receives and transmits the stream on *EndPoint* objects, which encapsulate the actual network connections. Each EndPoint has an associated thread that reads or writes data on the network, depending on the configuration of the EndPoint. A ControlThread object is responsible for managing the insertion, removal, and ordering of the filters associated with the stream. In this example, the proxy comprises three filters, F1, F2, and F3. The key support mechanisms are detachable stream objects, namely, DetachableInputStream (DIS) and DetachableOutputStream (DOS). The DIS and DOS are used for all communication among filters and between filters and EndPoints. The DIS and DOS can be stopped (paused), disconnected, and reconnected, enabling the dynamic redirection and modification of data streams. The I/O stream abstraction provides a convenient way to separate adaptive behavior from the application and other parts of the middleware. Now, let us briefly describe the main classes that make up the proxy infrastructure.

3.1 DetachableOutputStream and DetachableInputStream

These classes are implemented as modifications to the Java PipedOutputStream and PipedInputStream classes, respectively. DetachableOutputStream extends the base java.io.OutputStream class and DetachableInputStream extends the base java.io.InputStream class. In addition to overriding many of the base class methods, we have also included additional state variables and methods to implement the functionality needed to support composable filters. Fig. 4 illustrates the relationship between a DetachableOutputStream (DOS) and a DetachableInput Stream (DIS). The design is similar to that of Piped I/O streams. The connect() method is used to associate a specific output stream with a specific input stream. Among other tasks, the DOS.connect() method sets

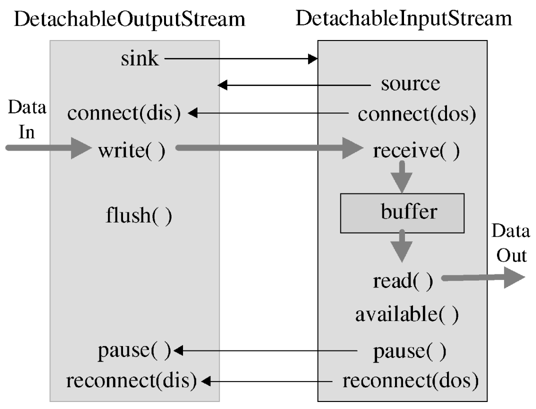


Fig. 4. Configuration of detachable streams.

DOS.sink and DIS.source variables so as to identify the other half of the connection. The DIS.connect() method simply calls the DOS.connect() method, which actually does this initialization.

As with their piped counterparts, the data written to the stream is buffered at the DIS side. An invocation of the DOS.write() method results in a call to the DIS.receive() method, which places the data in the buffer. The DIS.available() method returns the number of bytes currently in the buffer, and data is retrieved from the buffer using the DIS.read() method. The DOS.flush() method can be used to force any buffered output bytes to be written out and notifies any readers that bytes are waiting in the pipe.

Unlike the PipedOutputStream and PipedInputStream classes, the detachable stream classes include pause() and reconnect() methods, enabling the DOS and DIS to be temporarily stopped and reconnected to other DIS and DOS objects, respectively. The pause() method has to be called before disconnecting and switching the data stream. Code excerpted from DOS.pause() is given in Fig. 5. The method blocks attempts to write to the buffer and ensures that all the data has been read from the buffer. It also sets flags indicating that the two sides are no longer connected. An instance variable `swflag` is used to indicate that the stream is being paused and "switched." Once the pause method returns, the reconnect() method can be used to attach the DIS and DOS to other filters. If the buffer has not yet emptied, the caller is suspended until the buffer becomes empty. The reconnect() method checks whether the call is valid (not still in the connected state) and then mimics the actions of the connect() method in setting several global variables.

3.2 ControlThread

This class is used to manage the configuration of filters on a given stream supported by the proxy. The class maintains the Filter Vector, a dynamic array that holds references to the currently configured filters. The class implements methods to insert and remove filters from the Filter Vector, as well as methods that allow the ControlManger class to query about the available filters and the methods they support. The ControlThread receives commands from across the network, either from the mobile client, from an application server, or from the control manager (discussed later). The code segment in Fig. 6 is excerpted from the

```
public synchronized void pause(){

  // stop the DOS side
  if (this.swflag==false){
    this.swflag=true;
    this.connected=false;
  }

  // wait for data to clear
  if(!sink.checkBuf()){
    synchronized(sink.syncObj){
      try{
        sink.syncObj.wait();
      }catch(InterruptedException I)
        {System.err.println("Intr Wait!:" + I);}
    }

  // now stop the DIS side
  synchronized(sink){
    if( sink.swflag==false){
      sink.swflag=true;
      sink.connected=false;
      }
    }
  }
}
```

Fig. 5. Excerpt from DIS/DOS pause() method.

insert() method, which inserts a filter at an indexed location in a running stream.

3.3 Filter and FilterContainer

The base class for proxylets is Detachable.Proxylet. Any proxylet that is to be used within the RAPIDware infrastructure needs to extend this base class. The Proxylet class extends the Thread class and thus is inherently runnable. The Filter class extends the base class and is meant to be further extended by any specific proxy filter used in the system; see Fig. 7. The author of a filter writes

```
// Add a filter F to the current
// Vector V at the position pos.
public synchronized void insert(Filter F, int pos){

// Leftfilter is the filter to precede F
Filter LeftFilter = (Filter)V.element(pos-1);

// RightFilter is the filter to follow F
Filter RightFilter= (Filter)V.element(pos);

// We need pause only the left filter's DOS.
// The right filter's DIS is automatically paused.
// Then we can connect DIS and DOS of F.
LeftFilter.DOS.pause();
LeftFilter.DOS.reconnect(F.DIS);
RightFilter.DIS.reconnect(F.DOS);

F.start();  // Start the new Filter

// Insert the new Filter into the Vector
// and update the filter vector map.
V.insertElement(F,pos);
mapUpdate();
}
```

Fig. 6. Excerpt from Control Manager insert() method.

```
public class Filter extends Proxylet implements Serializable, Cloneable {
    String idString = new String("NA/");  // the filter identifier

    // The DIS, which will be manipulated by the ControlThread
    public final DetachableInputStream DIS = new DetachableInputStream();

    // The DOS, which will be manipulated by the ControlThread
    public final DetachableInputStream DOS = new DetachableOutputStream();

    private int objid = -1;  // An object id for convenience

    // The setDIS and setDOS methods allow on to set up their own
    // references names to the actual DIS and DOS
    public DetachableInputStream setDIS(){
        return(this.DIS);    }

    public DetachableOutputStream setDOS(){
        return(this.DOS);    }
```

Fig. 7. Excerpt from the Filter class.

the functional code as the run() method of the filter. The Filter class contains a DIS and a DOS object, along with their corresponding standard references, called *DIS* and *DOS*. The ControlThread uses these references to manipulate the stream connections. A group of methods (e.g., setDIS, setDOS, getid) is used to establish references to the DIS and DOS in the filter code itself.

This architecture can be used to define many types of filters since the developer can include any type of processing by overriding the filter's run() method. Hence, filters can be defined for error control, compression, encryption, virus scanning, and so on. The main constraint in the Filter class is that the connections to other components use the DIS/DOS pipeline structure. (The Proxylet superclass is more general in that an instance of this class can include arbitrary connections to other components.) The FilterContainer class is used to hold a vector of Filter objects. The FilterContainer class has methods to provide the number of Filters available and an enumeration method to return a String enumeration of the Filter objects names.

3.4 EndPoints

These are extensions of Filters that are instantiated by the ControlThread to provide proxy input and output. If the I/O is network-based, then the EndPoint objects would be a EndPointSocketReader and EndPointSocketWriter. If the I/O is a nonnetwork stream, then we would use an EndPointStreamReader and EndPointStreamWriter. Each EndPoint contains an active thread that handles I/O to and from the proxy. Combined with the ControlThread, two EndPoints comprise a "null" proxy, that is, one that simply forwards data without modifying it. Upon insertion of a filter between the EndPoints, the stream is redirected through the new filter.

3.5 Sender and Receiver Filter Matching

Each Filter has associated with it a unique ID and, possibly, a unique peerID. The former is used to identify the filter and the latter, if present, is used to identify a peer filter on the other side of the communication channel. Given a filter that performs some processing on outgoing packets, its *peer filter* performs the reverse processing on incoming packets. For example, a Compressor filter on the sending end of a connection requires a DeCompressor filter on the receiving end. Each filter, together with its peer filter, has to comply with a generic protocol that defines the format and use of application-level headers. Specifically, each filter on the sending side of a connection (for example, at a proxy) adds a header to the packets before writing them to its DOS. The first field in the header identifies the peer filter needed at the receiver; the remaining fields are used internally by the filter and the peer. When a packet arrives at the receiving side, it is first delivered to an EndPoint object, where the peerID of the packet is checked. If the EndPoint is connected to a filter whose ID matches the peerID contained in the incoming packet, then the EndPoint will deliver the packet to the filter via its DOS. If a match does not occur, then the ControlThread is notified since either a filter is missing from the FilterContainer or the ordering of filters (and their peers) has changed. The ControlThread will insert, remove, or reorder the filters to handle this situation. As a packet traverses peer filters at the receiving side, each peer filter verifies that the next ID in the packet header matches the next filter in the pipeline. If not, again the ControlThread is notified of the situation and the required modification to the pipeline is carried out. Once a packet has been processed by all necessary peer filters, it is delivered to the application.

3.6 ControlManager

To test the behavior of RAPIDware filters and related components, we found it useful to develop an interactive administration program to monitor and manage RAPID-ware-based collaborative sessions. The ControlManager class has a Swing-based GUI designed for this purpose. Based on responses to queries, the ControlManager constructs a graphical representation of the state of the

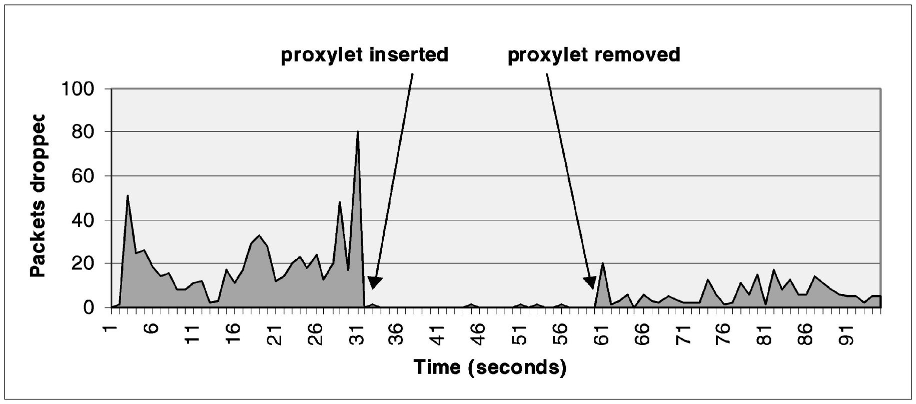


Fig. 8. Effect of dynamic insertion and removal of FEC filter.

proxy, including the current configuration of filters. The interface displays the current configuration of proxy filters and has pull-down menus and dialog text boxes that allow an administrator to insert and remove filters at specified locations in a composable proxy. Fig. 8 shows a sample trace in which the Control Manager was used to insert, then later remove, an FEC filter on an interactive audio stream being delivered to one of our wireless laptops. The reduction in packet loss rate is striking; details are discussed in Section 5.

We note that the latency for inserting filters is small, at most a few milliseconds. The actual switching of the data stream by reconfiguring DIS/DOS connections is negligible, a few instructions. The largest part of the potential delay is waiting for an internal DIS buffer to be read and processed by an existing (downstream) filter. However, by default, the DIS buffer size is only 1,024 bytes and, in our experience, the switching causes no noticeable gap in audio streams. Delays in delivering the data to the receiving application and, hence, limits on the number and type of filters that can be used together in a proxy, depend on a combination of factors: the amount of processing performed by each filter, the timing requirements of the data stream, and any filter-specific delay introduced at the receiving side. Section 4 discusses this issue in the context of audio streaming.

4 The Audio Filters and Their Composition

To evaluate the operation and performance of the RAPIDware DIS/DOS proxy infrastructure, as well as to determine which new features are needed, we have constructed several filters and other proxylets. For example, we have ported proxy services from Pavilion (such as video transcoding and reliable muliticasting services) to the new framework. We are also developing new proxylets related to security management and handoff of applications among different graphical displays. The focus of this paper is on our two different audio FEC filters, each of which can be independently inserted in a running audio stream in order to improve the quality of communication among mobile users. As we shall show later, chaining together the two filters can provide a level of error correction beyond what either filter can provide separately. We begin by discussing the packet loss characteristics of wireless LANs (WLANs) and their effects on interactive audio streaming, followed by details of the two audio FEC filters and a description of their combined operation.

4.1 Characteristics of Wireless LANs

The performance of group communication services, such as audio multicasting for collaborative Web applications, is affected by four main characteristics of WLANs. First, the packet loss rates are highly dynamic and location-dependent [20]. Fig. 9a demonstrates this behavior by plotting the relationship between signal-to-noise ratio (SNR) and packet loss rate during a short excursion within range of the wireless access point in our laboratory. The results demonstrate the highly variable loss rate that can occur in such environments as the SNR values quickly drop below the level of 20 dB that is typically considered acceptable.

Second, the loss characteristics of a WLAN are very different from those of a wired network. In a wired domain, losses occur mainly due to congestion and the subsequent buffer overflow. In the wireless domain, however, losses are more commonly due to external factors like interference, alignment of antennae, ambient temperature, and so on. Fig. 9b and Fig. 9c show example burst error distributions for two locations near our laboratory, where our wireless access point is located. Location 1 is just outside our laboratory and location 2 is approximately 25 meters down a corridor. In both cases, while some large bursts occur, many are very short and most "burst" errors comprise a single packet loss. To minimize the loss rate in terms of bytes, smaller packets are preferred, as shown in Fig. 9d. Apparently, the errors within larger packets are relatively localized so that, by sending several smaller packets, some number of them will be received successfully, whereas the larger packet would be lost.

Third, the 802.11b CSMA/CA MAC layer provides RTS/CTS signaling and link-level acknowledgments for unicast frames, but not for multicast frames. The result is a higher packet loss rate as observed by applications using UDP/IP multicast, as opposed to UDP unicast. Fig. 10 demonstrates this behavior for a typical location just outside our laboratory. Since multicast delivery of data streams is an inherent component of collaborative applications, error control on multicast data streams is a salient issue that needs to be addressed.

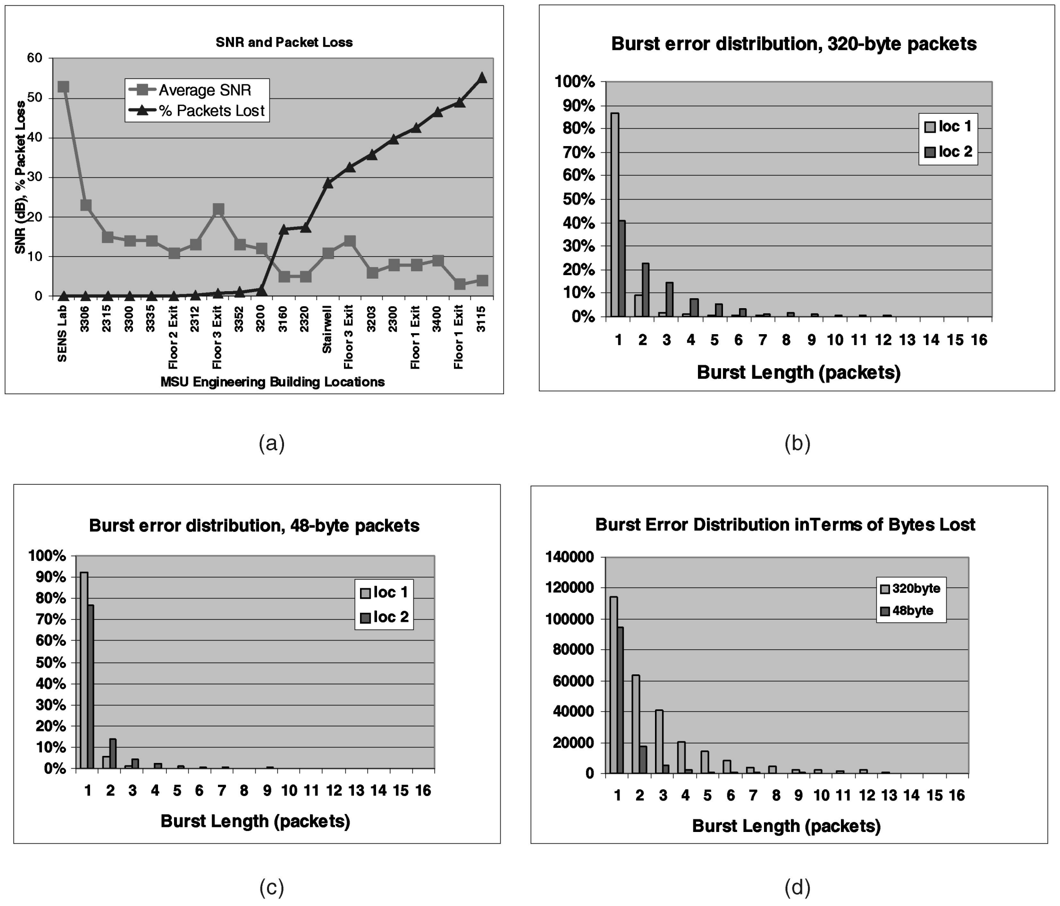


Fig. 9. Typical characteristics of a WLAN channel. (a) Packet loss versus SNR. (b) Burst distribution, 320-byte packets. (c) Burst distribution, 48-byte packets. (d) 320-byte versus 48-byte packets.

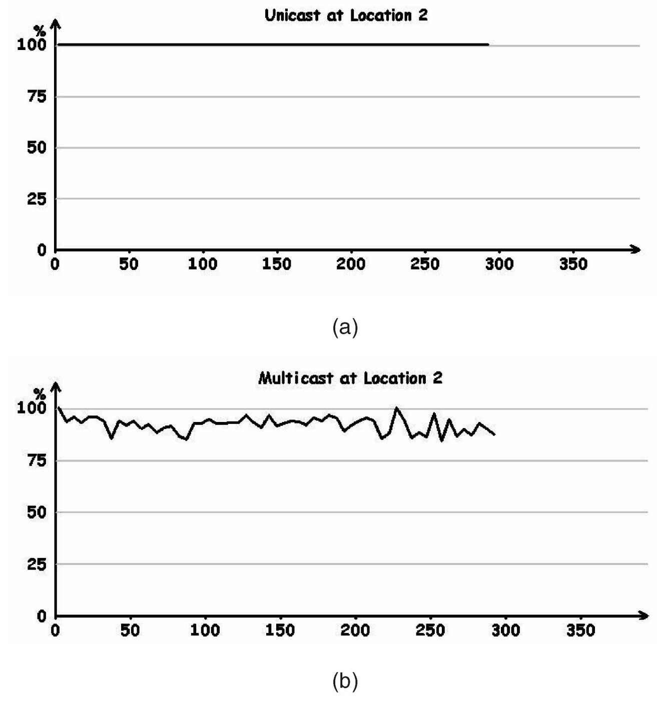


Fig. 10. Sample packet loss traces for UDP/IP unicast and multicast. (a) UDP unicast reception rate. (b) UDP/IP multicast reception rate.

Fourth, since the wireless channel is a shared broadcast medium, it is important to minimize the amount of feedback from receivers. Simultaneous responses from multiple receivers can cause channel congestion and burden the access point, thereby hindering the forward transmission of data. Thus, it is desirable to use proxy-based FEC instead of proxy-based retransmissions for real-time communication such as interactive audio streams.

4.2 Audio Filter Using Block-Oriented FEC

Our first audio filter uses an FEC mechanism that can be applied to any data type. It recovers packets that have been "erased" due to an error detected by the CRC check in the data link layer. As shown in Fig. 11, an (n, k) *block erasure code* converts k source packets into n encoded packets such that any k of the n encoded packets can be used to reconstruct the k source packets [32]. In this paper, we use only *systematic* codes, which means that the first k of the n encoded packets are identical to the k source packets. We refer to the first k packets as *data* packets, and the remaining $(n - k)$ packets as *parity* packets. Each set of n encoded packets is referred to as a *group*. The advantage of using block erasure codes for multicasting is that a single parity packet can be used to correct independent single-packet losses among different receivers [22]. These codes are lossless in that a successful decoding produces exactly the original data.

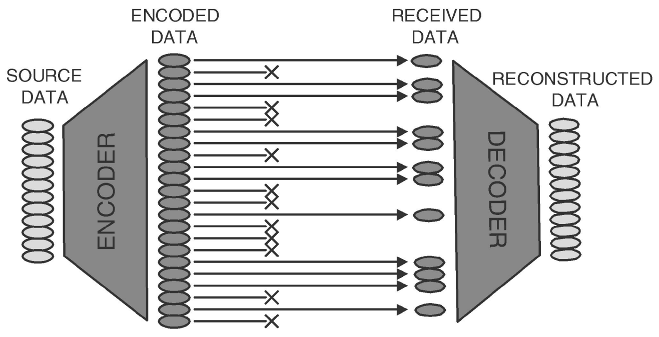


Fig. 11. Operation of FEC based on block erasure codes.

Recently, Rizzo [22] studied the feasibility of software encoding and decoding for packet-level FEC, using a particular block erasure code called the Vandermonde code. Depending on the values of k and $(n-k)$, Rizzo showed that this code can be efficiently executed on many common microprocessors. Rizzo's public domain FEC encoder/decoder library [22] is implemented in C and has been used in many projects involving multicast communication [33], [34], [35], [36], [37], including our own prior studies [20], [28], [29]. Given the emphasis on portability and code mobility in the RAPIDware project, however, we decided to switch to an open-source Java implementation of Rizzo's FEC library, available from Swarmcast [38]. In general, we found the Swarmcast library to provide a convenient interface and good performance. Although slower than the C implementation, the Java version (including native code) is able to satisfy real-time audio encoding and decoding requirements on systems with modest processing power.

Fig. 12 shows the operational schematic of the major components of the audio application when this FEC filter is running. The audio recorder was built as a pure Java application making use of the Java Sound API. Specifically, the recording thread uses the javax.sound package to read audio data from a workstation's sound card and send it to the proxy via the wired network. The encoding of the data is 16 bits per sample, PCM signed, at the standard rate of 8,000 Hz over a mono channel. The audio receiver uses one thread to read data from the network and store it in a circular buffer. A second thread reads the data and uses the Java Sound API to play it.

For data streams directed toward the client system, the encoder is instantiated on the proxy. The decoder on the client will be instantiated automatically after the arrival of the first FEC packet. The encoder and decoder filters have the same basic construction and simply invoke different methods in the Swarmcast FEC library. After creating the FEC encoder filter, the encoder thread loops in its run() method, collecting packets from the network. When k packets have been received in the filter's DIS, the thread invokes the encode() method, which returns n packets contained in a new reference buffer. These packets are labeled with a group identifier and sequence number and are written to the data stream. The decoder at the client requires reception of any k packets in a given group in order to decode the original k data packets. The decoder thread thus reads up to k packets in a given group, after which additional packets are discarded. This data is passed to the decode() method of the FEC codec, which returns the k original data packets, which are forwarded to the client application. As an optimization, if all the original k data packets arrive intact, then the decoder is bypassed. On the other hand, if fewer than k packets in the group arrive, then any data packets (among the first k packets) are forwarded to the application, while the remaining (parity) packets are discarded.

4.3 GSM Audio Filter

While block-oriented FEC approaches are effective in improving the quality of interactive audio streams on wireless networks [29], the group sizes must be relatively small in order to reduce playback delays. (In our studies, we typically use (n,k) values of (6, 4) or (8, 4).) Hence, the overhead in terms of parity packets is relatively high. An alternative approach with lower delay and lower overhead is *signal processing-based FEC (SFEC)* [39], [40], in which a lossy, compressed encoding of each packet i is piggybacked onto one or more subsequent packets. If packet i is lost, but

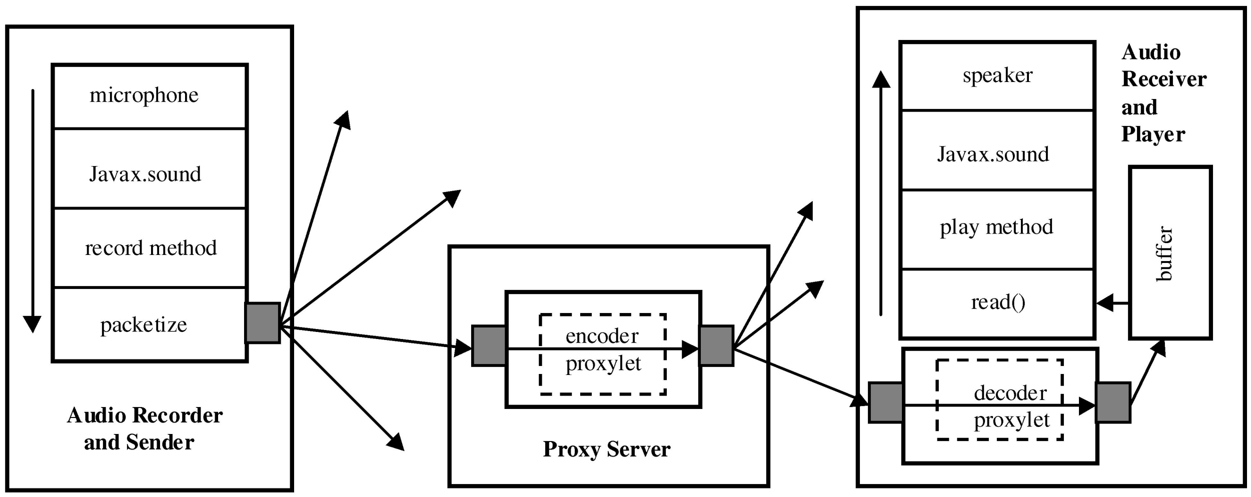


Fig. 12. Operational block diagram of the streaming audio configuration.

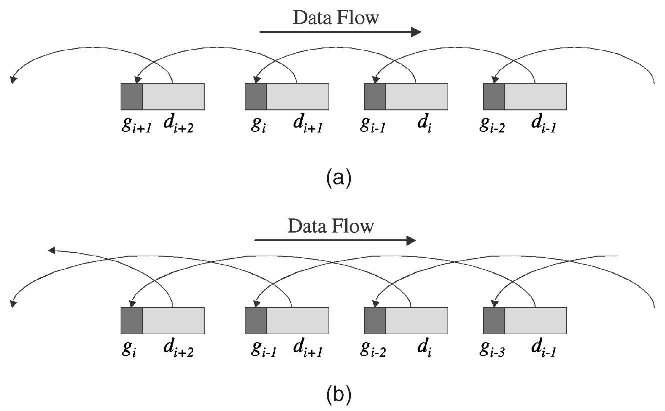

Data Flow
g_{i+1} d_{i+2} g_i d_{i+1} g_{i-1} d_i g_{i-2} d_{i-1}
(a)
Data Flow
g_i d_{i+2} g_{i-1} d_{i+1} g_{i-2} d_i g_{i-3} d_{i-1}
(b)

Fig. 13. Different ways of using GSM encoding on a packet stream. (a) GSM encoding with $\theta = 1$. (b) GSM encoding with $\theta = 2$.

one of the encodings of packet i arrives at the receiver, then at least a lower quality version of the packet can be played to the listener. The parameter θ is the offset between the original packet and its compressed version. Fig. 13 shows two different examples, one with $\theta = 1$ and the other with $\theta = 2$. As mentioned, it is also possible to place multiple encodings of the same packet in the subsequent stream, for example, using both $\theta = 1$ and $\theta = 3$.

The RAPIDware filter that we developed uses GSM 06.10 encoding [23] for generating the redundant copies of packets. The full rate speech codec in GSM is described as Regular Pulse Excitation with Long Term Prediction (GSM 06.10 RPE-LTP). Although GSM is a CPU-intensive coding algorithm—it is 1,200 times more costly than normal PCM encoding [40]—the bandwidth overhead is very small. Specifically, the GSM encoding creates only 33 bytes for a PCM-encoded packet containing up to 320 bytes (160 samples). Our filter uses the Tritonus Java version of the GSM codec, a freeware package available under GNU public license.

The GSM filter can work in one of two ways. The first is to piggyback encoded data on subsequent packets, as shown in Fig. 13. The second is to compute encodings for *multiple* packets and create a new packet of encoded data, to be inserted in the stream at a later point. Fig. 14 shows an example in which the GSM encodings on each of three packets are combined into a new packet that is inserted after the next group of three packets. This second method is useful when it is important to keep packet sizes small, as in a wireless LAN.

4.4 Combining the Audio Filters

Either a GSM or FEC filter can be inserted separately into an audio stream. However, we can also insert both filters, as

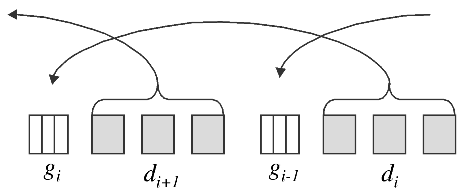


Fig. 14. Application of GSM filter to three-packet groups.

shown in Fig. 15 (in the figure and in the remainder of the paper, we will refer to the filters simply as "GSM" and "FEC"). If the direction of the audio channel were reversed, then the encoders would reside on the client and the decoders on the proxy.

Fig. 16 shows a particular example of the two encoders working together. The GSM encoder is configured to operate on a packet basis, computing a GSM encoding for every three data packets and inserting the new packet after $3 \times \theta$ data packets in the following packet stream. Each group of four packets (three data packets and one GSM packet) is forwarded to the FEC filter, which is configured to compute two parity packets using a (6, 4) block erasure code. The (6, 4) code can recover up to two lost packets per group, hence covering the most common packet loss cases, and does so without any loss in quality. Combining FEC and GSM code can tolerate relatively long isolated burst errors, depending on the location of the lost packets relative to group boundaries. Of course, if GSM instead of FEC is used to reconstruct a packet, the quality of the resulting signal will be lower than that of the original.

Use of FEC introduces bandwidth overhead that depends on the values of n, k, and θ. Considering the example illustrated in Fig. 16, if the size of the data packets is 100 bytes, then the overhead is approximately 100 percent (three packets of error correcting information for every three packets of data). While this rate seems relatively high, the 128 kbps rate of our audio channels is low compared to many other types of traffic. Moreover, a single multicast audio channel serves multiple participants in a collaborative Web session. In such sessions, maintaining an effective audio channel among the users is perhaps more important than the quality of service of other data types, such as streaming video.

Another issue important to real-time communication is the additional delay introduced into the packet stream. While processing at the proxy introduces a small delay, our experience indicates that stream-specific processing delays at the receiver are more significant. For example, let us consider the use of (8, 4) FEC audio encoding in which each packet contains 3 milliseconds of live audio data. Assume

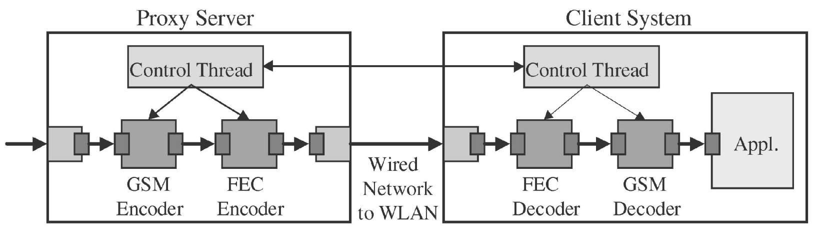


Fig. 15. Configuration of audio filters on proxy and client.

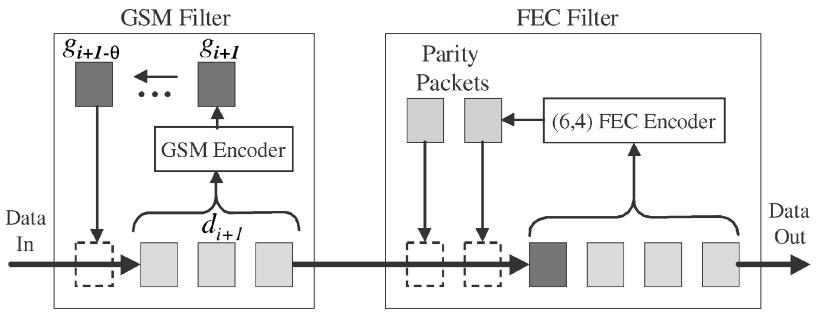


Fig. 16. Operation of combined audio filters.

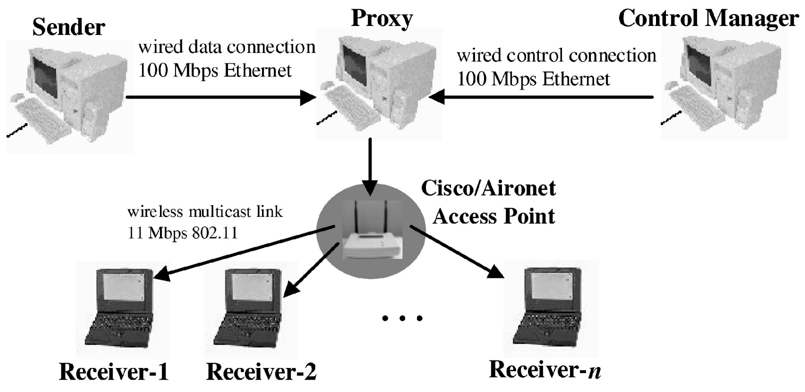


Fig. 17. Physical configuration of experimental components.

that all four data packets are lost and all four parity packets arrive at the receiver. Decoding cannot begin until the fourth parity packet arrives. Even if the proxy can send the four parity packets immediately following the data packets (which have a natural spacing of 3 milliseconds) and encoding and decoding are instantaneous, the playback of decoded audio will be delayed by at least 9 milliseconds. The values of n, θ, and the packet payload size must be chosen so that the playback delay does not seriously affect audio quality.

5 Experimental Evaluation

In order to study the operation and performance of audio filters separately and in combination, we conducted a set of experiments on the mobile computing testbed in our Software Engineering and Network Systems (SENS) Laboratory.

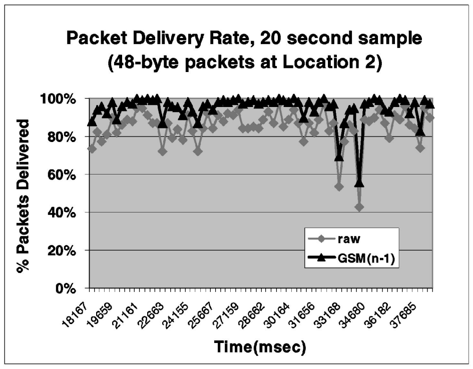
5.1 Testing Environment

The mobile testbed includes conventional workstations connected by a 100 Mbps Fast Ethernet switch, three 802.11 WLANs (Lucent WaveLAN, Proxim RangeLAN2, and Cisco/Aironet), and several mobile handheld and laptop computer systems. All tests reported here were conducted on the Aironet WLAN, which uses direct sequence spread spectrum signaling and has a raw bit rate of 11 Mbps. We used both wired desktop PCs and wireless laptop PCs as participating stations in the experimental configuration depicted in Fig. 17. The sender, proxy, and control manager were executed on dual-processor 400/450 MHz desktop workstations, while the mobile nodes were 300 MHz laptops equipped with Aironet network interface cards. The Aironet access point and the participating wired stations were located in our laboratory, while the locations of the mobile nodes were varied. Although the proxy multicasts the audio stream on the WLAN, here we report the results for a single receiver.

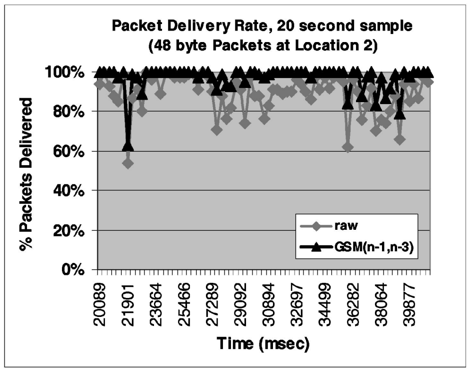
Initially, both the proxy and client are configured as "null" filters, as the Endpoints simply read and retransmit data. In an actual RAPIDware environment, observer threads at the client would monitor the packet loss rate and burst length distribution and would inform the ControlThread on the proxy of the current situation. The ControlThread decides when to insert the FEC or GSM filter. In order to control the testing, however, we used the Control Manager GUI to insert the filters manually.

5.2 Experimental Results

We started by testing the GSM filter in isolation, setting θ to different values and using both single and double copies of the encoded data. In all cases, small 48-byte packets produced considerably better results than 320-byte packets, so we report only the former here. Fig. 18 shows a sample of the results. In Fig. 18a, we placed a single encoded copy of each packet i in its successor packet $i+1$. In Fig. 18b, we placed one encoded copy in packet $i+1$ and one in packet $i+3$. Using multiple copies produces a clear advantage in terms of packet delivery rate. The bandwidth overhead rates are 69 percent and 138 percent, respectively, when considering only payload bytes. When including MAC, IP, and UPD headers and MAC-layer gaps and preamble bytes (66 bytes total per packet), the rates drop to 29 percent and 58 percent since this method introduces only payload bytes, but no new packets.

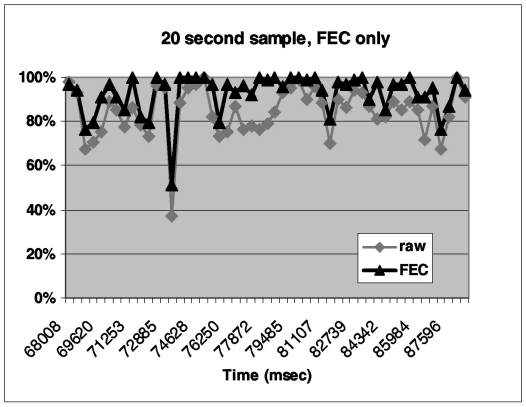


(a)

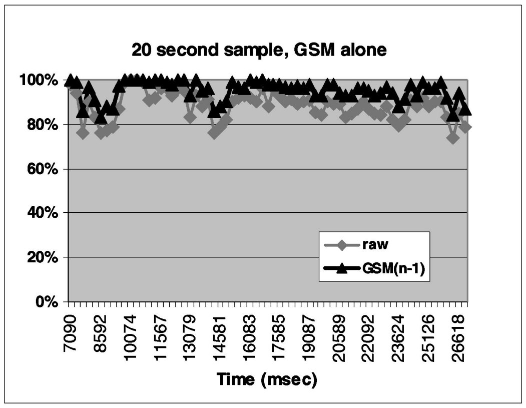


(b)

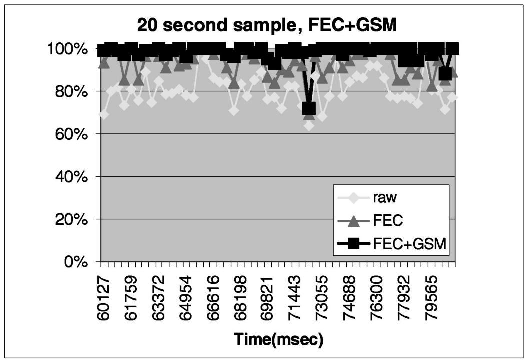
Fig. 18. Sample traces results for the GSM filter with 48-byte packets. (a) $\theta = 1$, 48-byte packets. (b) $\theta = 1, 3$, 48-byte packets.



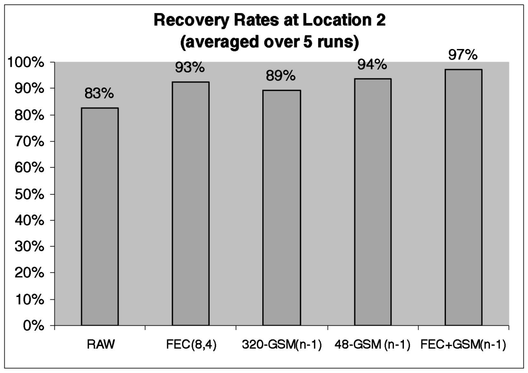
(a)



(b)



(c)

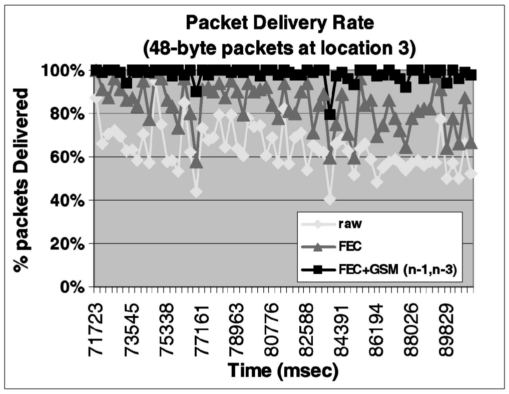


(d)

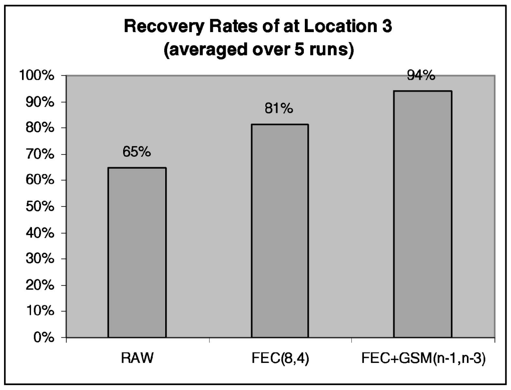
Fig. 19. Effects of FEC, GSM filters, and their composition. (a) FEC(8, 4) alone. (b) GSM(n-1) along. (c) FEC(8, 4)+GSM(n-1). (d) Performance comparision.

Fig. 19a, Fig. 19b, and Fig. 19c, respectively, show sample traces of using the FEC(8, 4) and GSM(n-1) filters, alone and in combination. When used alone, the overhead for FEC is 100 percent since this code doubles the number of packets transmitted. The GSM code in these tests uses 33-bytes to encode three 48-byte packets, so the overhead is $33/144 = 30\%$. The overhead of the combination is $(4 * 48 + 33)/(3 * 48) = 156\%$. The combination is most effective in recovering data and this result is confirmed by Fig. 19d, where we averaged the results of five two-minute runs and computed the packet delivery rate for each of the methods at a particular location in our building. By combining the two filters, we are able to reconstruct 97 percent of the audio data, even though the raw packet delivery rate at this location was only 83 percent.

Finally, we conducted a set of experiments near the periphery of the wireless cell (Location 3), where large burst errors are more frequent. In these tests, we again combined the FEC filter with the GSM filter, but we configured the latter to use two values of θ, both 1 and 3, which enables it to correct longer burst errors. The overhead of this combination is $(4 * 48 + 2 * 33)/(2 * 48) = 269\%$. Fig. 20a



(a)



(b)

Fig. 20. Effects of FEC and GSM filters near the wireless cell periphery. (a) Sample trace. (b) Performance comparison.

shows a short sample trace. Despite the fact that the raw delivery rate sometimes falls below 50 percent, the combination of filters is extremely effective in recovering the lost data. Fig. 20b shows the average results of five two-minute runs. At this location, the raw delivery rate was only 65 percent, FEC alone raised the rate to 81 percent, and the GSM filter further improved the rate to 94 percent. The 13 percent GSM improvement over FEC alone compares with only a 4 percent improvement at Location 2. We conclude that, while both types of filters improve the quality of the audio channel, near the cell periphery, they are almost equally important. These results demonstrate the utility of being able to compose two different proxylets easily and dynamically within a single proxy framework.

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 CEA [1], MOOSCo [2], BRAIN [3], Squirrel [4], Adapt [5], MASH [9], TAO [10], MobiWare [11], MCF [12], QuO [13], MPA [8], Odyssey [14], and DaCapo++ [15], Rover [6], BARWAN [30], and Sync [41]. These projects have greatly improved the understanding of how middleware can accommodate device heterogeneity and dynamic network conditions, particularly in the area of adaptive communication protocols and services. Indeed, several projects address dynamic configuration of proxies. In the remainder of this section, we discuss four such projects and their relationship to the work presented here.

Zenel [17] developed a general-purpose proxy system that enables filters to be downloaded and inserted dynamically on various types of data streams. Different filters are available for different data streams (MPEG, HTTP, TCP, and so forth). Zenel conducted extensive experiments that demonstrated the usefulness of stream-specific filters. Both high-level filters (above the socket layer) and low-level filters (requiring kernel support) are supported. While the implementation is not described in detail, Zenel does note relatively large insertion delays. The DIS/DOS mechanism described herein is intended to provide a simple and perhaps more flexible way to reconfigure user-level proxy filters and, as such, complements Zenel's work. In particular, we hide proxy reconfiguration within the I/O stream abstraction in order to separate adaptive behavior from nonadaptive behavior. Also, the switching to newly inserted filters requires only a few machine instructions.

In the MobiWare project [11], "mobile filters" can be dispatched to various nodes in the network, or to hosts, in order to achieve bandwidth conservation. Apparently, these filters are established only during handoff from one network to another. The detachable stream infrastructure discussed herein could be used to extend this functionality so that filters could be reorganized at any time.

The Adapt project at Lancaster [5] uses open bindings to support manipulation and reconfiguration of communication paths. The associated object graph mechanism could be used directly to implement dynamically composable proxy services through composition of meta-objects. In contrast to the use of compositional reflection, in this project, we sought to determine the minimal level of functionality needed to provide dynamic composition of communication stream components. The advantage of implementing adaptive behavior within existing low-level communication mechanism, such as detachable Java I/O streams, is that developers can construct application code and core middleware services without changing their methodology. The adaptive part of the middleware can be developed separately. This strategy also facilitates the porting of legacy code to a new environment that requires adaptability.

The Berkeley TranSend proxy is based on the TACC model [7] in which *workers* are the active components of the proxy. The TACC server enables workers to be chained together in a manner similar to Unix pipes. Details of the implementation are not available. However, the project focuses on proxies built atop highly available parallel workstation clusters, whereas RAPIDware proxies are intended to be lightweight, on-demand proxies established dynamically on one or more idle workstations available to the user.

The Stanford Mobile People Architecture (MPA) [8] is designed to support person-to-person reachability through the use of *personal proxies*. A key component of the personal proxy is the use of *conversion drivers*, which are configured dynamically to match the capabilities of the user's device and network. The RAPIDware project complements this work by developing programming abstractions to support the design of such software. Specifically, the detachable stream mechanism and filter container class could be used to compose MPA drivers and facilitate their dynamic loading and unloading from across the network.

Finally, we emphasize that this paper has described only a small part of the RAPIDware project. A given external event, such as a sudden decrease in quality on a wireless link, can affect not only communication protocols, but also middleware components associated with fault tolerance, security, and user interfaces. The overall goal of the RAPIDware project is to develop an integrated methodology and programming language support for middleware adaptability that accommodates cross-cutting concerns in multiple dimensions. We will report developments in these areas in future papers.

7 Conclusions and Future Work

In this paper, we have described the use of our detachable Java I/O stream framework to support composition of proxy services. We presented the design of the framework, which enables proxylets to be dynamically inserted, removed, and reordered without disturbing existing network connections. We then demonstrated the use of the framework to support an important component of mobile collaborative computing, namely, the use of forward error correction to improve the quality of multicast audio streams. We developed two different audio FEC filters, one using block erasure codes and the other using the GSM 06.10 encoder, inserted them into the framework, and evaluated their performance on a WLAN testbed. The main contribution of this work is to show that independent proxylets can be composed easily and can cooperate in a synergistic manner, given the proper supporting proxy framework.

Our continuing work in this area addresses several issues: developing additional proxylets for the RAPIDware framework, developing a rules engine to characterize the "composability" of proxylets, and application of RAPIDware concepts to intrusion detection, fault tolerance, and user interfaces handoff. Given the increasing presence of wireless networks in homes and businesses, we envision application of the proposed techniques to improve performance of collaborative applications involving users who roam within a wireless environment.

7.1 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.

Acknowledgments

The authors would like to thank Robel Barrios, Aaron Malenfant, Jesus Arango, Ji Li, Arun Mani, Suraj Gaurav, Peng Ge, and Chiping Tang for their contributions to this work. This work was supported in part by the US Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by US National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724. This work was conducted while U.I. Padmanabhan and N. Ancha were graduate students at Michigan State University.

References

[1] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic Support for Distributed Applications," *Computer*, vol. 33, no. 3, pp. 68-76, 2000.

[2] H. Miranda, M. Antunes, L. Rodrigues, and A.R. Silva, "Group Communication Support for Dependable Multi-User Object-Oriented Environments," *Proc. SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Oct. 2000.

[3] L. Burness, A. Kassler, P. Khengar, E. Kovacs, D. Mandato, J. Manner, G. Neureiter, T. Robles, and H. Velayos, "The BRAIN Quality of Service Architecture for Adaptable Services," *Proc. Int'l Symp. Personal, Indoor, and Mobile Radio Comm. (PIMRC 2000)*, Sept. 2000.

[4] T. Kramp and R. Koster, "A Service-Centered Approach to QoS-Supporting Middleware (Work-in-Progress Paper)," *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Sept. 1998.

[5] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "A Software Architecture for Adaptive Distributed Multimedia Applications," *IEE Proc.—Software*, vol. 145, no. 5, pp. 163-171, 1998.

[6] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek, "Mobile Computing with the Rover Toolkit," *IEEE Trans. Computers*, special issue on mobile computing, vol. 46, no. 3, Mar. 1997.

[7] A. Fox, S.D. Gribble, Y. Chawathe, and E.A. Brewer, "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Comm.*, Aug. 1998.

[8] M. Roussopoulos, P. Maniatis, E. Swierk, K. Lai, G. Appenzeller, and M. Baker, "Person-Level Routing in the Mobile People Architecture," *Proc. 1999 USENIX Symp. Internet Technologies and Systems*, Oct. 1999.

[9] 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," *Proc. Seventh Int'l Workshop Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[10] 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," *Proc. IFIP Sixth Int'l Workshop Protocols for High-Speed Networks (PfHSN '99)*, Aug. 1998.

[11] 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 Comm. Magazine*, special issue on adapting to network and client variability, Aug. 1998.

[12] B. Li and K. Nahrstedt, "A Control-Based Middleware Framework for Quality of Service Adaptations," *IEEE J. Selected Areas in Comm.*, vol. 17, Sept. 1999.

[13] 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," *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Sept. 1998.

[14] B.D. Noble and M. Satyanarayanan, "Experience with Adaptive Mobile Applications in Odyssey," *Mobile Networks and Applications*, vol. 4, pp. 245-254, 1999.

[15] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A Flexible Middleware for Multimedia Communication: Design Implementation, and Experience," *IEEE J. Selected Areas in Comm.*, vol. 17, pp. 1580-1598, Sept. 1999.

[16] B.R. Badrinath, A. Bakre, R. Marantz, and T. Imielinski, "Handling Mobile Hosts: A Case for Indirect Interaction," *Proc. Fourth Workshop Workstation Operating Systems*, Oct. 1993.

[17] B. Zenel, "A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment," *Wireless Networks*, vol. 5, pp. 391-409, 1999.

[18] L. Chen and T. Suda, "Designing Mobile Computing Systems Using Distributed Objects," *IEEE Comm. Magazine*, vol. 35, Feb. 1997.

[19] Y. Chawathe, S. Fink, S. McCanne, and E. Brewer, "A Proxy Architecture for Reliable Multicast in Heterogeneous Environments," *Proc. ACM Multimedia '98*, Sept. 1998.

[20] P.K. McKinley and A.P. Mani, "An Experimental Study of Adaptive Forward Error Correction for Wireless Collaborative Computing," *Proc. IEEE 2001 Symp. Applications and the Internet (SAINT-01)*, Jan. 2001.

[21] L. Yang and M. Hofmann, "OPES Architecture for Rule Processing and Service Execution," Internet Draft draft-yang-opes-rule-processing-service-execution-00.txt, Feb. 2001.

[22] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," *ACM Computer Comm. Rev.*, Apr. 1997.

[23] J. Degener and C. Bormann, "The GSM 06.10 Lossy Speech Compression Library and Its Applications," 2000, available at http://kbs.cs.tu-berlin.de/jutta/toast.html.

[24] P.K. McKinley, A.M. Malenfant, and J.M. Arango, "Pavilion: A Distributed Middleware Framework for Collaborative Web-Based Applications," *Proc. ACM SIGGROUP Conf. Supporting Group Work*, pp. 179-188, Nov. 1999.

[25] P.K. McKinley, R.R. Barrios, and A.M. Malenfant, "Design and Performance Evaluation of a Java-Based Multicast Browser Tool," *Proc. 19th Int'l Conf. Distributed Computing Systems*, pp. 314-322, 1999.

[26] J. Arango and P.K. McKinley, "VGuide: Design and Performance Evaluation of a Synchronous Collaborative Virtual Reality Application," *Proc. IEEE Int'l Conf. Multimedia and Expo*, July 2000.

[27] P.K. McKinley and J. Li, "Pocket Pavilion: Synchronous Collaborative Browsing for Wireless Handheld Computers," *Proc. IEEE Int'l Conf. Multimedia and Expo*, July 2000.

[28] P. Ge and P.K. McKinley, "Experimental Evaluation of Error Control for Video Multicast over Wireless LANs," *Proc. Third Int'l Workshop Multimedia Network Systems*, Apr. 2001.

[29] P.K. McKinley and S. Gaurav, "Experimental Evaluation of Forward Error Correction on Multicast Audio Streams in Wireless LANs," *Proc. ACM Multimedia 2000*, pp. 416-418, Nov. 2000.

[30] R.H. Katz et al., "The Bay Area Research Wireless Access Network (BARWAN)," *Proc. Spring COMPCON Conf.*, 1996.

[31] P.K. McKinley and U.I. Padmanabhan, "Design of Composable Proxy Filters for Mobile Cmputing," *Proc. Second Int'l Workshop Wireless Networks and Mobile Computing*, Apr. 2001.

[32] A.J. McAuley, "Reliable Broadband Communications Using Burst Erasure Correcting Code," *Proc. ACM SIGCOMM*, pp. 287-306, Sept. 1990.

[33] L. Rizzo and L. Vicisano, "RMDP: An FEC-Based Reliable Multicast Protocol for Wireless Environments," *ACM Mobile Computer and Comm. Rev.*, vol. 2, Apr. 1998.

[34] J. Nonnenmacher, E.W. Biersack, and D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission," *IEEE/ACM Trans. Networking*, vol. 6, no. 4, pp. 349-361, 1998.

[35] C. Huitema, "The Case for Packet Level FEC," *Proc. IFIP Fifth Int'l Workshop Protocols for High-Speed Networks (PfHSN '96)*, pp. 110-120, Oct. 1996.

[36] J. Gemmell, E. Schooler, and R. Kermode, "A Scalable Multicast Architecture for One-to-Many Telepresentations," *Proc. IEEE Int'l Conf. Multimedia Computing Systems*, pp. 128-139, 1998.

[37] R. Kermode, "Scoped Hybrid Automatic Repeat ReQuest with Forward Error Correction (SHARQFEC)," *Proc. ACM SIGCOMM*, Sept. 1998.

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

[39] M. Podolsky, C. Romer, and S. McCanne, "Simulation of FEC-Based Error Control for Packet Audio on the Internet," *Proc. IEEE INFOCOM '96*, Mar. 1998.

[40] J.-C. Bolot and A. Vega-Garcia, "Control Mechanisms for Packet Audio in Internet," *Proc. IEEE INFOCOM '96*, pp. 232-239, Apr. 1996.

[41] J. Munson and P. Dewan, "Sync: A System for Mobile Collaborative Applications," *Computer*, vol. 30, no. 6, pp. 59-66, 1997.

Philip K. McKinley received the BS degree in mathematics and computer science from Iowa State University in 1982, the MS degree in computer science from Purdue University in 1983, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1989. He is currently a professor in the Department of Computer Science and Engineering at Michigan State University. He was previously a member of technical staff at Bell Laboratories. Dr. McKinley is an associate editor for the *IEEE Transactions on Parallel and Distributed Systems* and is cochair of the program committee for IEEE ICDCS 2003. His current research interests include adaptive middleware, collaborative applications, mobile computing, and group communication protocols. He is a member of the IEEE and the IEEE Computer Society.

Udiyan I. Padmanabhan received the MS degree in computer science from Michigan State University in 2002. He is a program manager in the Windows CE Operating System group at Microsoft Corporation in Redmond, Washington. His current interests include embedded operating systems, mobile devices, and distributed middleware technologies.

Nandagopal Ancha received the MS degree in computer science from Michigan State University in 2001. He is currently a software engineer with Ericsson IP Infrastructure located in Raleigh, North Carolina. His current interests are in the design and analysis of communcation systems and studying complexities involved from the perspective of an edge router.

Seyed Masoud Sadjadi received the BS degree in hardware computer engineering from the University of Tehran in 1995, the MS degree in software computer engineering from Azad Tehran University in 1999. He is currently a doctoral student in the Department of Computer Science and Engineering at Michigan State University. His current research interests include adaptive software, adaptive middleware, collaborative applications, mobile computing, multimedia application, and programming languages. He is a student member of the IEEE and the IEEE Computer Society.