MetaSockets: Run-Time Support for Adaptive Communication Services

S. M. Sadjadi, P. K. McKinley, and E. P. Kasten

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
{sadjadis,mckinley,kasten}@cse.msu.edu

(Extended Abstract)

Introduction. Rapid improvements in mobile computing devices and wireless networks promise to provide a foundation for ubiquitous computing. However, distributed software must be able to adapt to dynamic situations related to several cross-cutting concerns, including quality-of-service, fault-tolerance, energy management, and security. We are currently conducting a project called *RAPIDware* that addresses the design of adaptive software for dynamic, heterogeneous environments. We previously introduced Adaptive Java, an extension to the Java programming language, which provides language constructs and compiler support for the development of adaptive software. This extended abstract describes the use of Adaptive Java to develop an adaptable communication component called the MetaSocket. MetaSockets are created from existing Java socket classes, but their structure and behavior can be adapted at run time in response to external stimuli. Experiments demonstrate how MetaSockets respond to dynamic wireless channel conditions to improve the quality of interactive audio streams delivered to iPAQ handheld computers.

Adaptive Java. Adaptive Java [1] is rooted in computational reflection, which refers to the ability of a computational process to reason about and possibly alter its own behavior. The basic building blocks of an Adaptive Java program are *components*. The key programming concept in Adaptive Java is to provide each component with three types of interfaces: one for performing normal imperative operations on the object (*invocations*), one for observing internal behavior (*refractions*), and one for changing internal behavior (*transmutations*). An existing Java class is converted into an adaptable component in two steps. In the first step, a *base-level* Adaptive Java component is constructed from a Java class through an operation called *absorption*, which uses the `absorbs` keyword. 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. Adaptive Java is currently implemented using a source-to-source compiler that translates the new language extensions into Java.

MetaSocket Design and Implementation. To explore the ability of Adaptive Java for support of run-time adaptability, we used this language to design and implement a "metamorphic" socket (MetaSocket) component. Fig. 1 depicts 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. Fig. 1(a) depicts a Java MulticastSocket class and a subset of its public methods: receive(), send(), close(), joinGroup(), and leaveGroup(). Fig. 1(b) shows a SendMSocket component, which is designed to be used as a *send-only* multicast socket. The SendMSocket component *absorbs* the Java multicast socket class and implements send() and close() invocations that can be used by other components. Other methods of the base-level class are occluded. A link between an invocation and a method indicates a dependency. For example, the send() invocation depends on the send() method, because its implementation calls that method. Fig. 1(c) shows a MetaSendMSocket component, which metafies an instance of the SendMSocket component. A refractive MOP includes a getStatus() method and a transmutative MOP includes insertFilter() and removeFilter() methods. The use and operation of these MOPs will be explained shortly. In a similar manner, a *receive-only* MetaSocket can be created for use on the receiving side of a communication channel. In addition to receive() and close() invocations, the RecvMSocket base-level component provides joinGroup() and leaveGroup() invocations, which are needed for joining and leaving an IP multicast group.

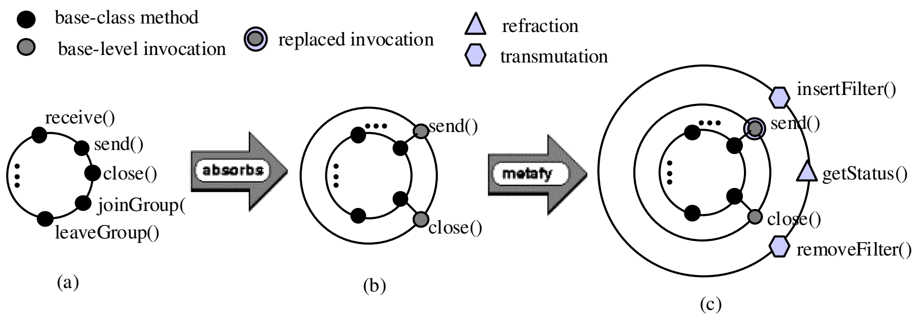

Fig. 1. MetaSocket absorption and metafication. (a) Java MulticastSocket; (b) SendMSocket component; (c) MetaSendMSocket meta-level component.

Fig. 2 illustrates the internal architecture of both a MetaSendMSocket and a MetaRecvMSocket, as configured in our study. Packets are passed through a pipeline of Filter components, each of which processes the packets. Filters interact through PacketBuffer components. 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 our implementation, when a packet is processed by a filter on the sender side, an filter header may be prepended to the packet. On the receiver, these headers identify the processing order and filters required to reverse the transformations applied by the sender.

Case Study. We conducted a case study in which we used MetaSockets to enhance interactive audio streaming over wireless channels to iPAQ handheld computers. Filters were constructed to measure and report packet loss and im-

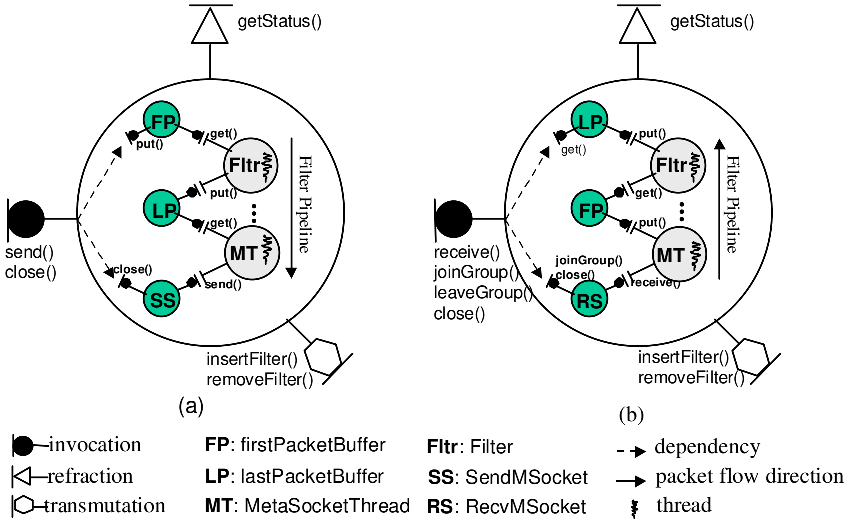

Fig. 2. MetaSocket architecture: (a) MetaSendMSocket; (b) MetaRecvMSocket.

plement forward error correction codes. The framework is event driven. EventMediator (EM) components are used to decouple event generators from event listeners, and DecisionMaker (DM) components are used to control the nonfunctional behavior of Adaptive Java components. Fig. 3 shows an example trace that plots packet loss as observed by two different loss monitoring filters on the receiver. The Network Packet Loss curve shows two periods of high packet loss. The Application Packet Loss curve shows the effect of dynamic insertion and removal of an FEC filter. 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 an upper threshold (30%). The filter fires an *UnAcceptableLossRateEvent*, causing the local DM to request an FEC filter. A global DM decides to insert two filters in the MetaSendMSocket filter pipeline: an FECEncoder that uses (8,4) block encoding, and a SendNetLossDetector filter that monitors raw packet loss rate. When packets containing the headers of the two new filters begin arriving at the receiver, the RecvAppLossDetector fires two *FilterMismatchEvent* events, which cause a RecvNetLossDetector filter and a FECDecoder filter to be inserted in the MetaRecvMSocket filter pipeline.

As shown in Fig. 3, the $(8,4)$ FEC code is very effective at 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 a 10% lower threshold, so it fires an *AcceptableLossRateEvent*, which results in the removal of the FEC filter by the global DM. 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 on the receiver, and the peer filters are removed. As a result, the loss rate experienced by the application is again identical to the network loss rate. At packet set 60, the FEC filter is again inserted, due to high loss rate, and it is later removed at packet set 80.

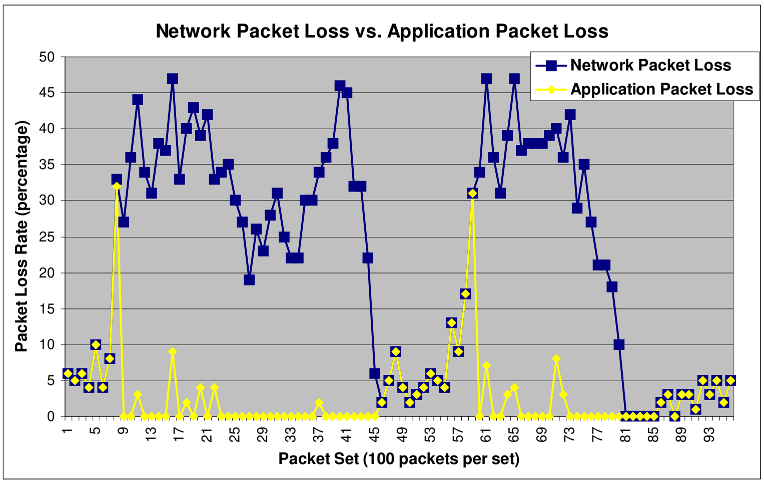

Fig. 3. MetaSocket packet loss behavior will dynamic filter insertion.

Considering Fig. 3 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 nonfunctional concerns to be added to the system after it is already deployed and executing. Complete details of the MetaSocket architecture and the case study, as well as discussion of related work, can be found in [2]. The RAPIDware project homepage is `http://www.cse.msu.edu/rapidware`.

Acknowledgements. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by U.S. National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724.

References

1. Kasten, E.P., McKinley, P.K., Sadjadi, S.M., Stirewalt, R.E.K.: Separating introspection and intercession in metamorphic distributed systems. In: Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02), Vienna, Austria (2002)
2. Sadjadi, S.M., McKinley, P.K., Kasten, E.P.: Metasockets: Run-time support for adaptive communication services. Technical Report MSU-CSE-02-22, Department of Computer Science, Michigan State University, East Lansing, Michigan (2002)